

6.205

Finite State Machines

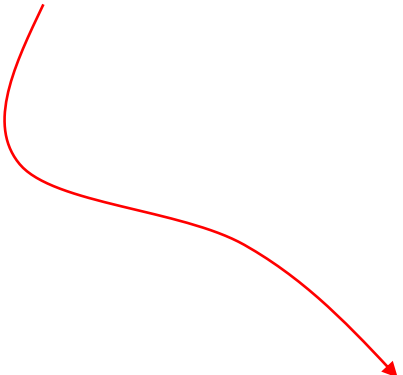
Administrivia

- Week 2 due last night.
- Week 3 out after class today

Cool/Not Cool Bug

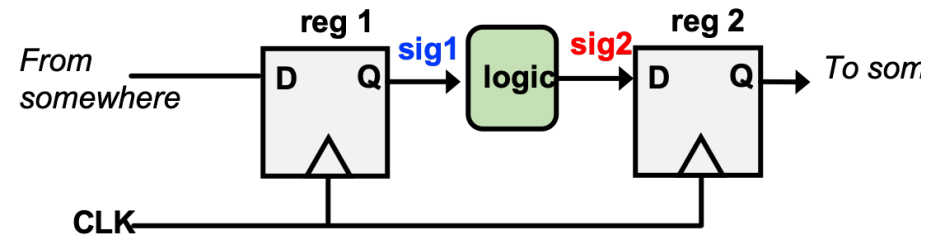
- What's wrong with this

```
module simple_counter( input wire clk,  
                        input wire rst,  
                        input wire evt,  
                        output logic[15:0] count  
                        );  
    always_ff @(posedge clk) begin  
        if (rst) begin  
            count <= 16'b0;  
        end else if (evt) begin  
            count <= count + 1;  
        end  
    end  
endmodule
```



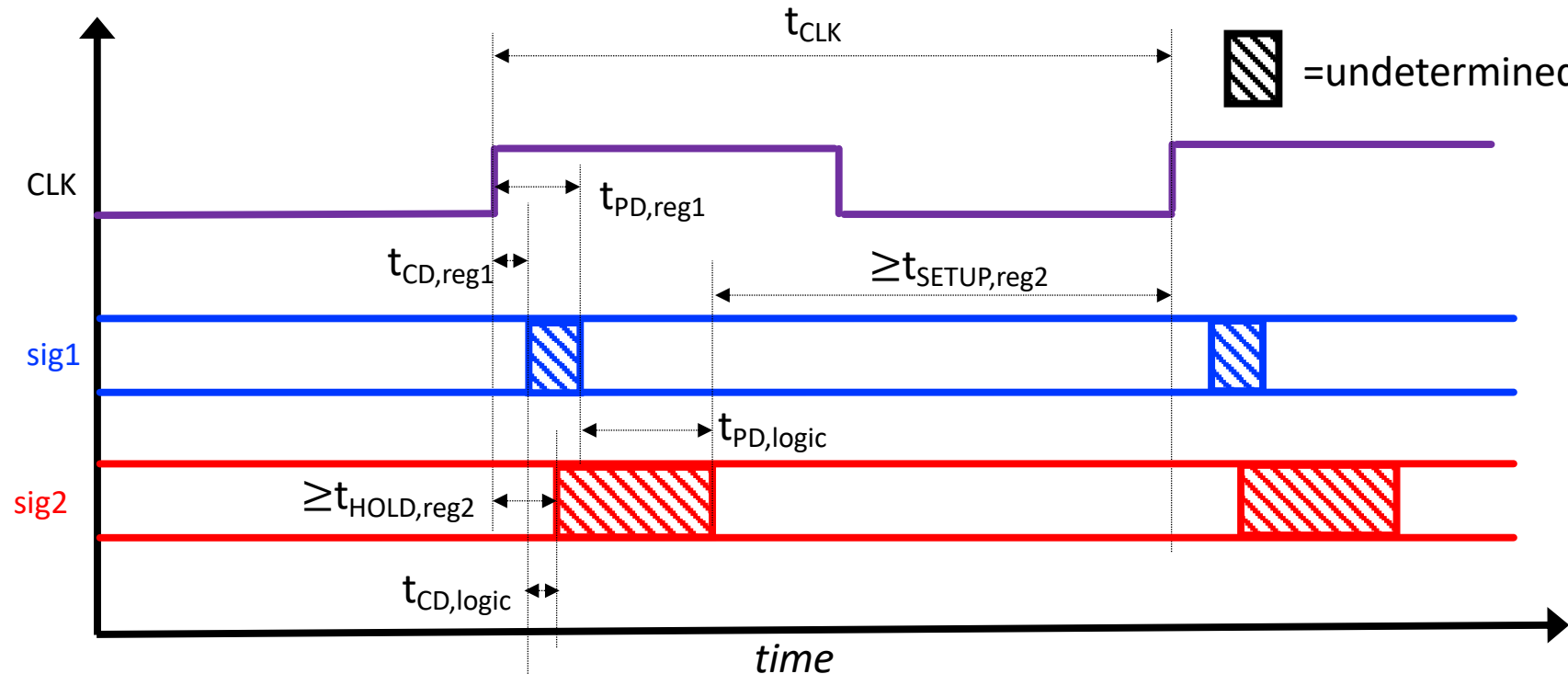
```
simple_counter msc(  
    .clk(clk),  
    .rst(counter_reset),  
    .evt(clk),  
    .count(count));
```

Register-to-Register Timing



— =determined state

▨ =undetermined state



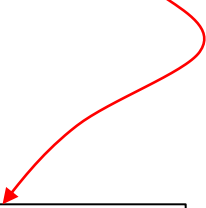
**Two Requirements/
Conclusions:**

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

Clocks Stay with Clocks!

- And Signals stay with signals. None of this stuff:



```
always_ff @(posedge some_button_thing)begin
    x <= 5+x;
    //other code
end
```

This will not play nice with other circuits that are clocked off an actual clock!

Interfacing to Sequential Logic

...Or what are the problems with working with Sequential Logic?....

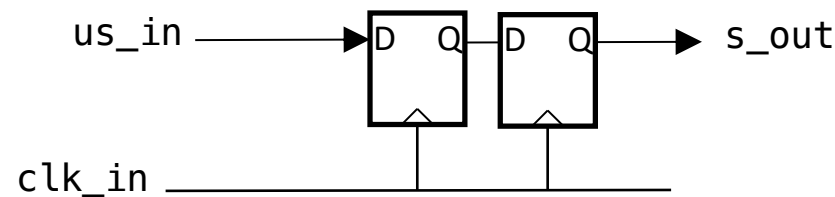
Huh?

- In Week 3:
- You'll be using these occasionally... or an equivalent...
- They build this
- Obviously that's a typo
- Right??????

```
module synchronizer #(parameter SYNC_DEPTH = 2)
    ( input wire clk,
      input wire rst,
      input wire us_in, //unsync_in
      output logic s_out); //sync_out

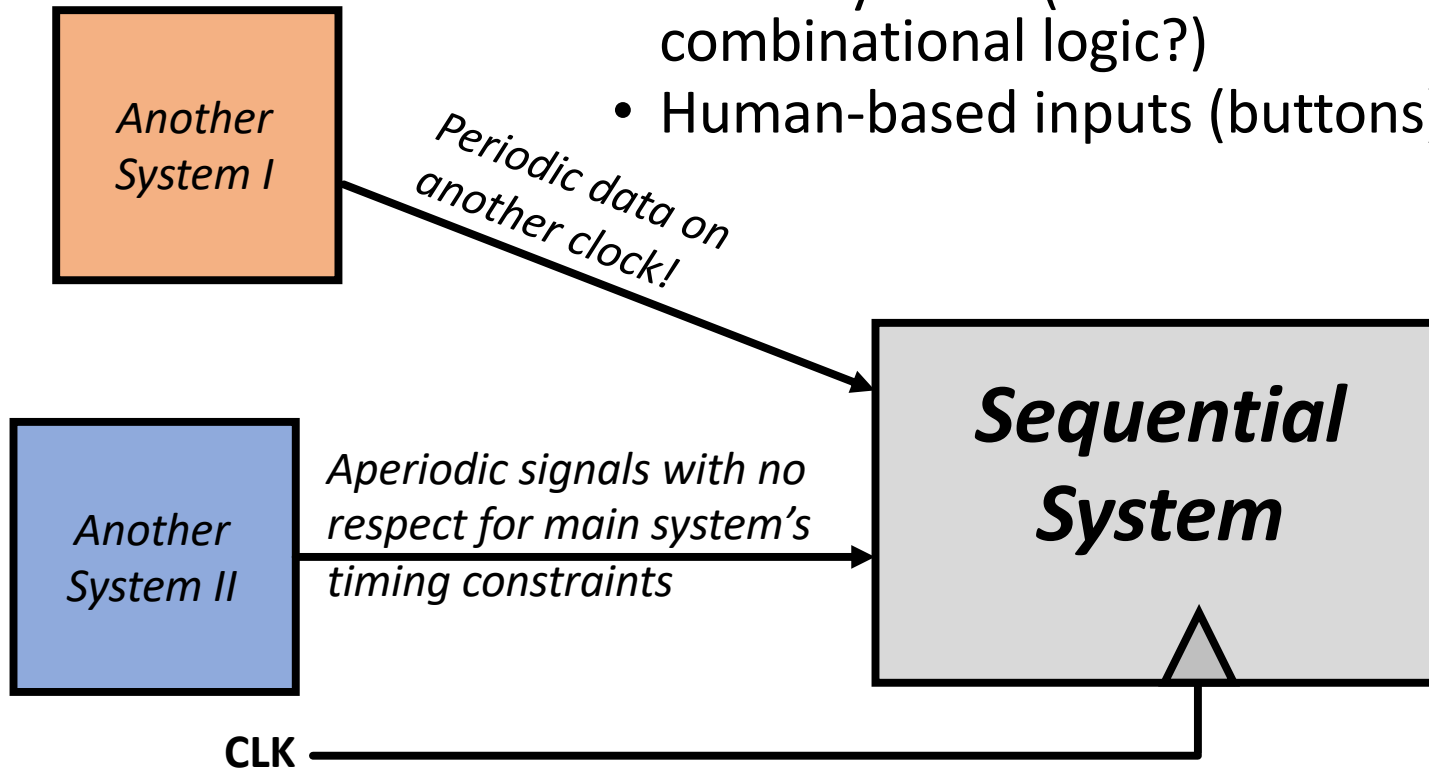
    logic [SYNC_DEPTH-1:0] sync;

    always_ff @(posedge clk)begin
        if (rst)begin
            sync <= {(SYNC_DEPTH){us_in}};
        end else begin
            sync[SYNC_DEPTH-1] <= us_in;
            for (int i=1; i<SYNC_DEPTH; i= i+1)begin
                sync[i-1] <= sync[i];
            end
        end
    end
    assign s_out = sync[0];
endmodule
```



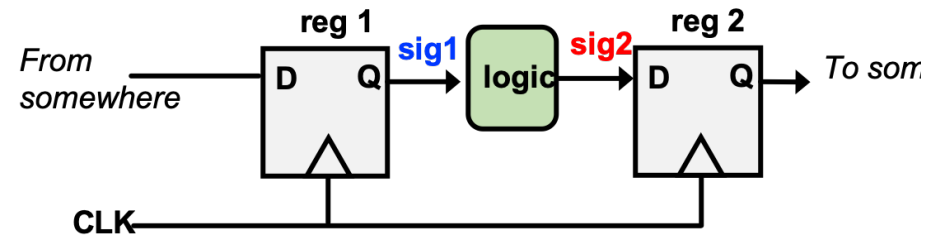
What if...?

- ...we need to interface with outside equipment:
 - Other systems (on different clocks or from combinational logic?)
 - Human-based inputs (buttons)



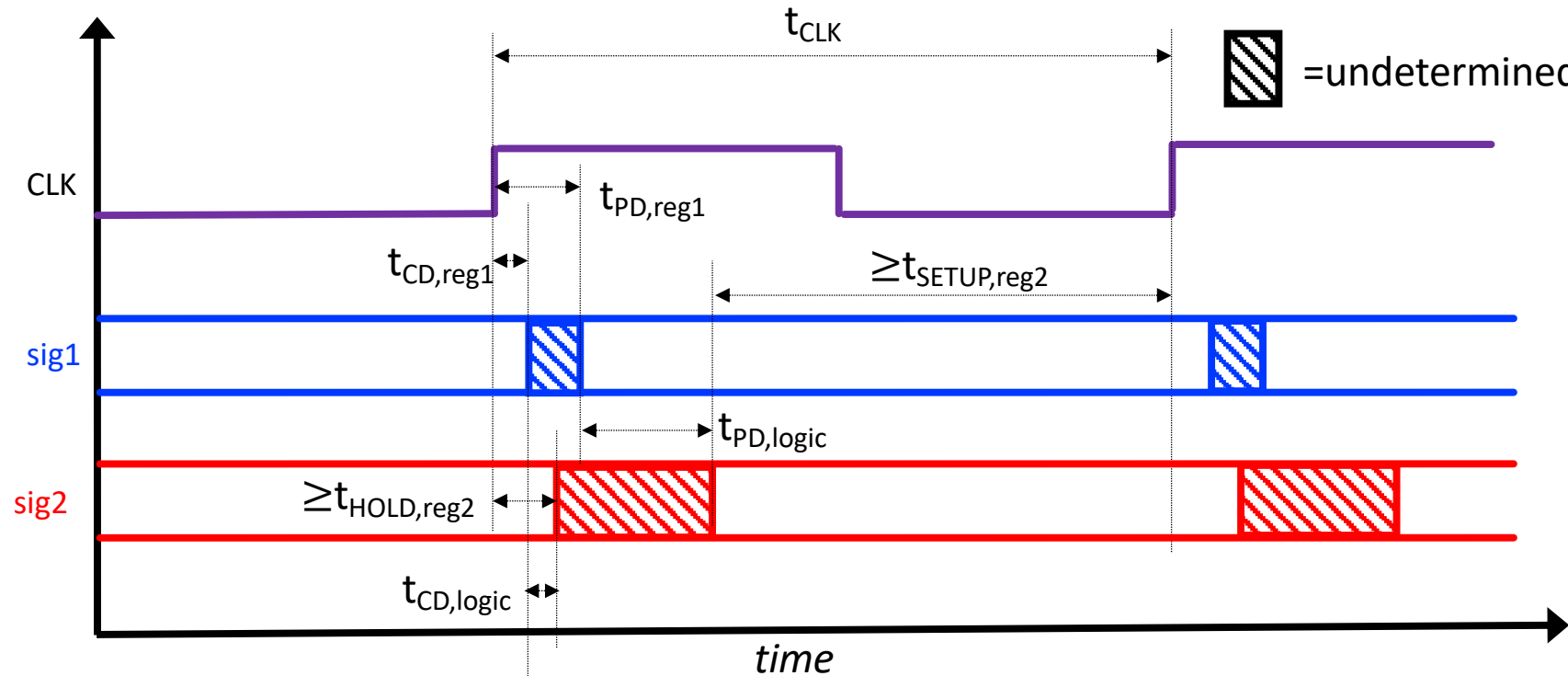
Can't guarantee setup and hold times will be met!

Register-to-Register Timing



— =determined state

▨ =undetermined state



**Two Requirements/
Conclusions:**

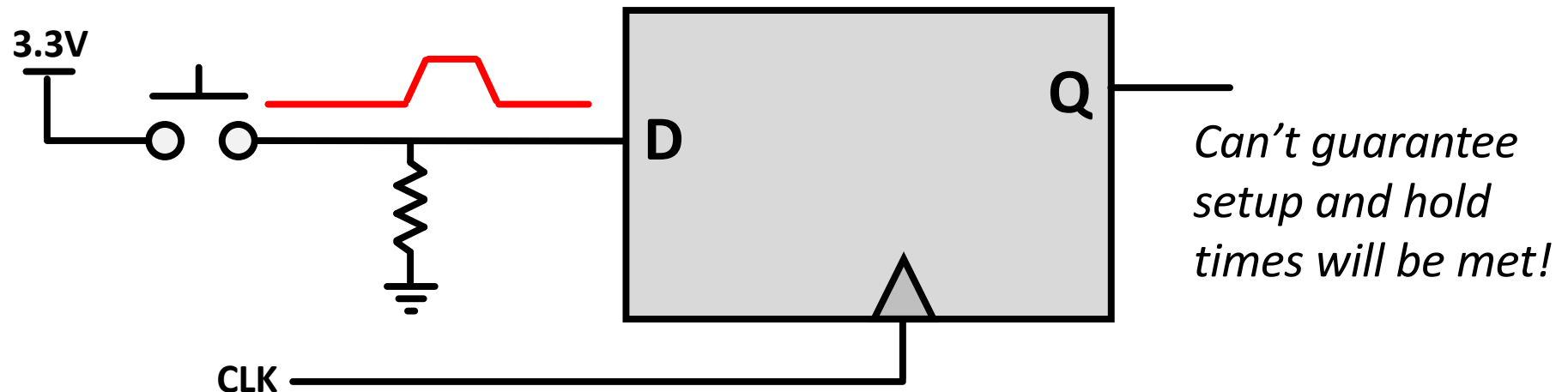
$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

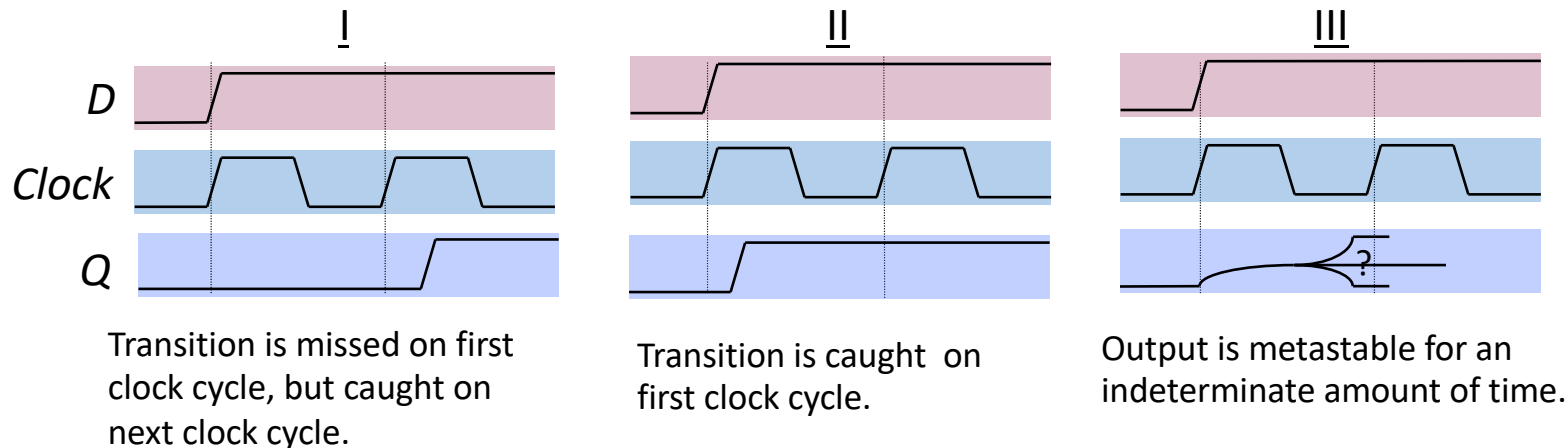
Single Clock Synchronous Discipline

- The timing requirements are already complicated enough with one clock. Avoid multiple clocks at all cost! DO NOT clock flip flops on non-clock lines.
- Single Clock signal shared among all clocked devices (one clock domain)
- Only care about the value of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Change saved state after noise-inducing logic changes have stopped!

Example: Asynchronous Inputs in Sequential Systems



When an asynchronous signal causes a setup/hold violation...



Q: Which cases are problematic?

Metastability

- D-registers have issues with all that feedback and stuff going on. Can go **metastable**
- Metastability is where the system hovers between Logic High and Logic Low in an unpredictable way

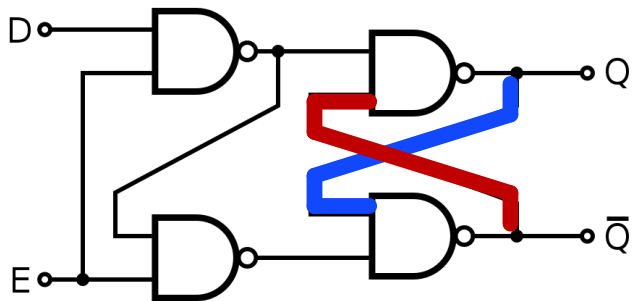


Figure 2. Effects of Violating t_{SU} & t_H Requirements

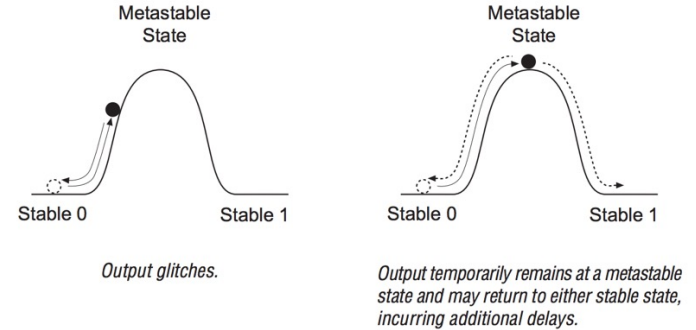
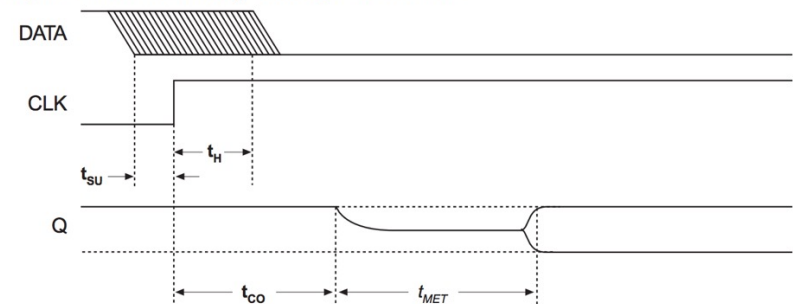


Figure 1. Metastability Timing Parameters



Metastability in Altera (®) Devices
Altera Application Note 42 (1999)

t_{CO} = "min time from clock to output"
....think of it as t_{pd} here (not exactly the same)

Handling Metastability

- Can't globally prevent metastability, but can isolate it!
- Stringing several registers together can isolate any freakouts!

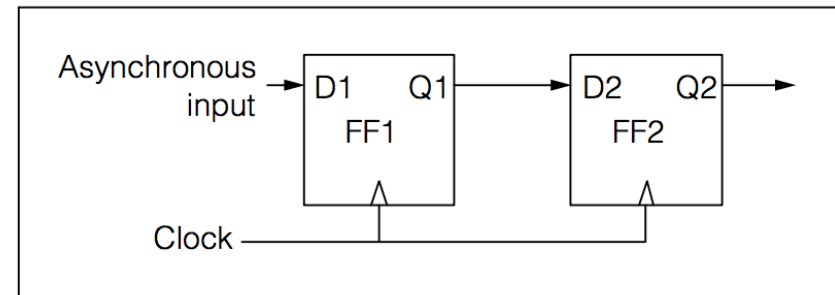
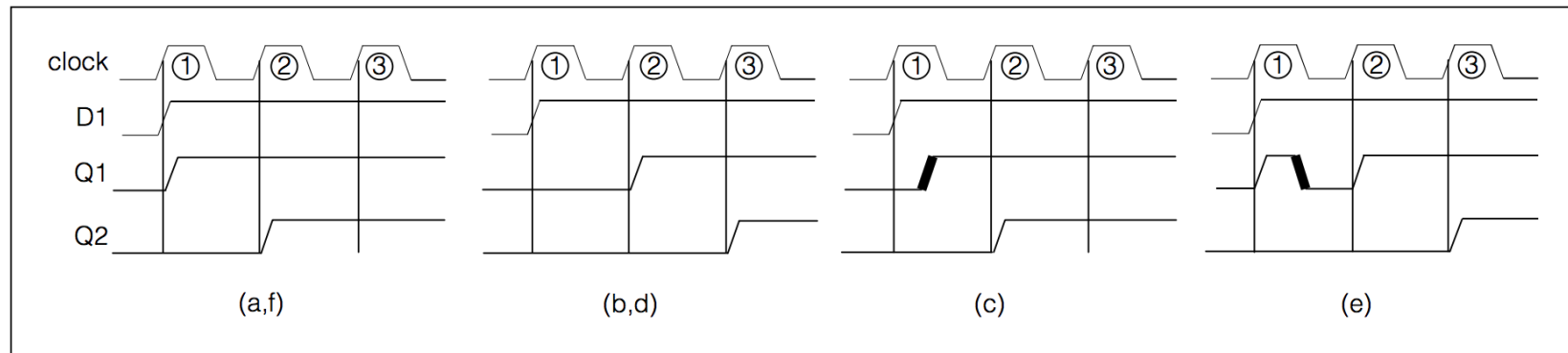


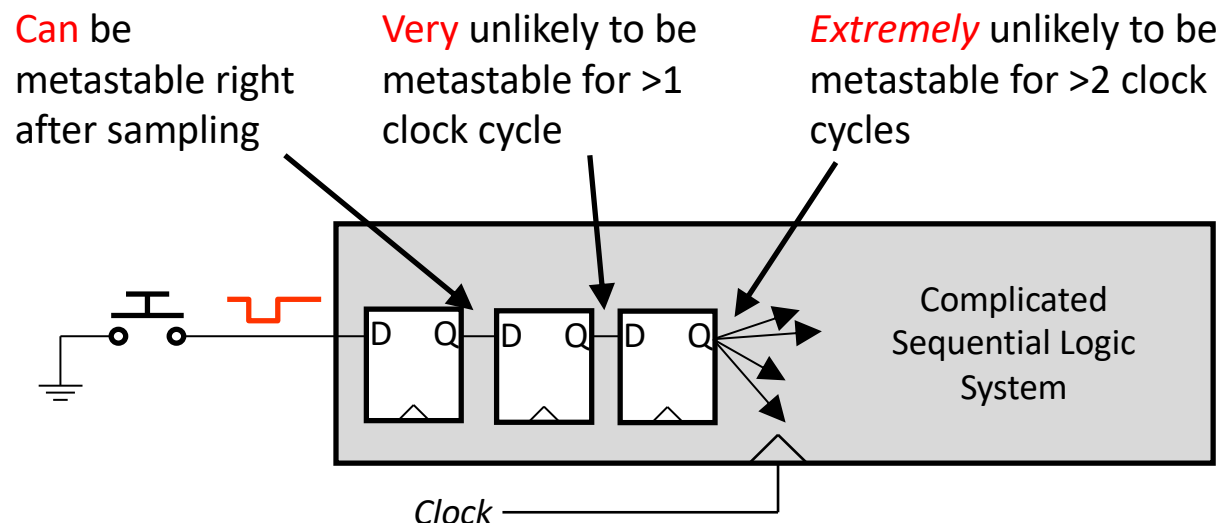
Figure 8. Two-flip-flop synchronization circuit.



“Metastability and Synchronizers: A Tutorial”
Ran Ginosar, Technion Israel Institute of Technology

Handling Metastability

- Completely preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize

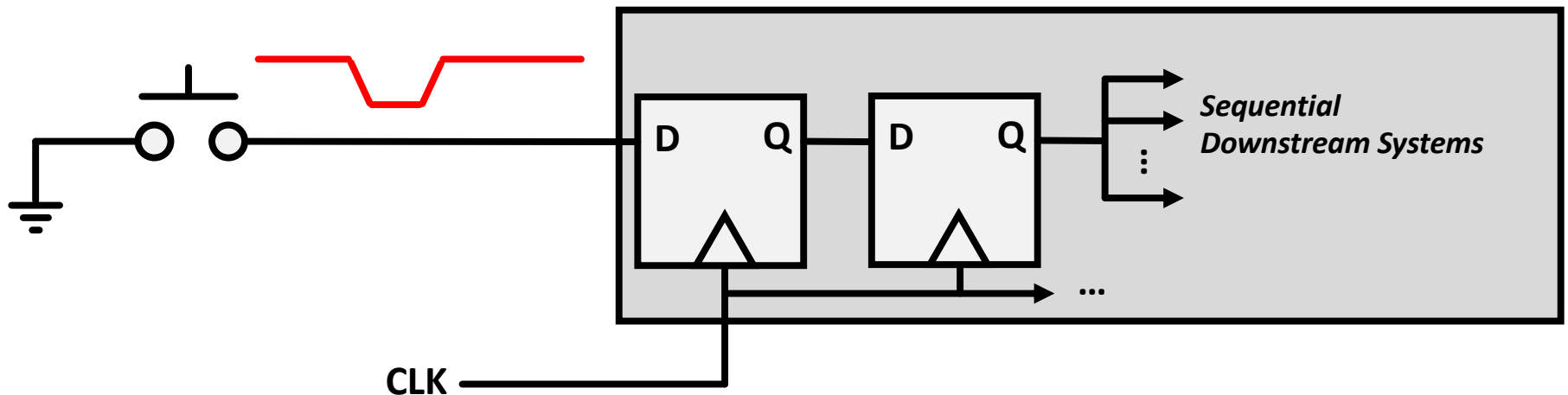
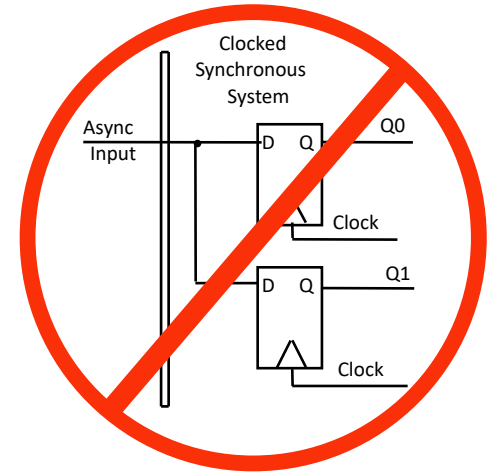


How many registers are necessary in 6.205?

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.205, a **pair of synchronization** registers is sufficient
- And for simple designs...with low t_{pd} you may not even need anything

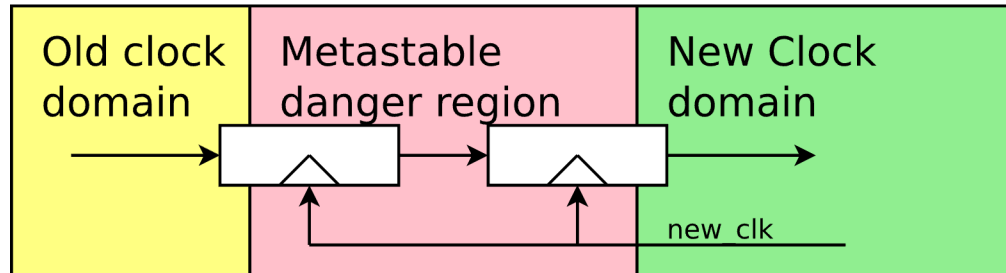
Handling Metastability

- Don't break off an asynchronous input until it has gone through some registers
- Basically: Ensure that external signals feed exactly one flip-flop chain before branching



Related: Clock Domain Crossing

- For example:
 - Data gets sent in at 25 MHz from one device (running on its own clock)
 - Your system runs at 50 MHz



```
1 //xfer_pipe can be >2 bits wide (2 is usually fine...3 better)
2 always_ff @(posedge new_clock)
3   { new_val, xfer_pipe } <= { xfer_pipe, i_val };
```

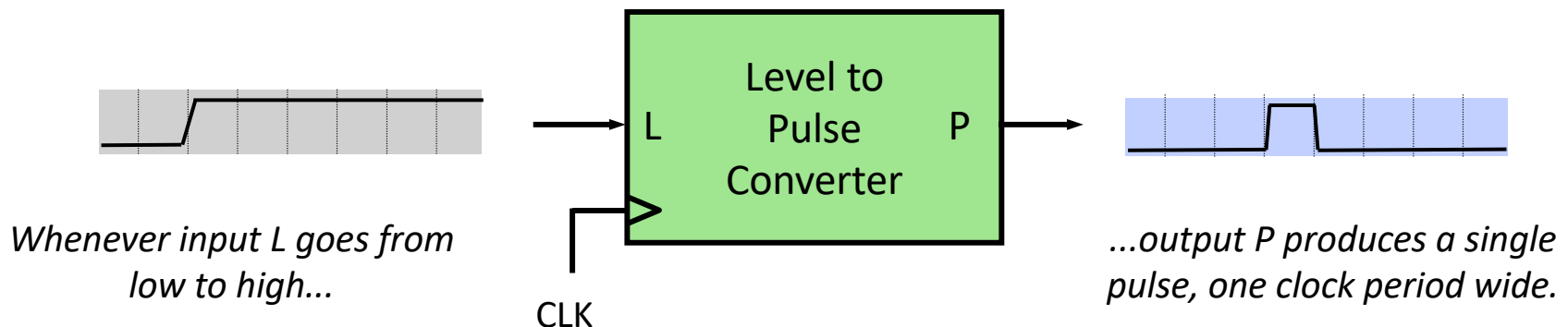
- This only works when original clock domain frequency is less than or equal to new clock domain frequency

<https://zipcpu.com/blog/2017/10/20/cdc.html>

State Machines

Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a **synchronous** rising-edge detector.
- Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for modules



Level-to-Pulse

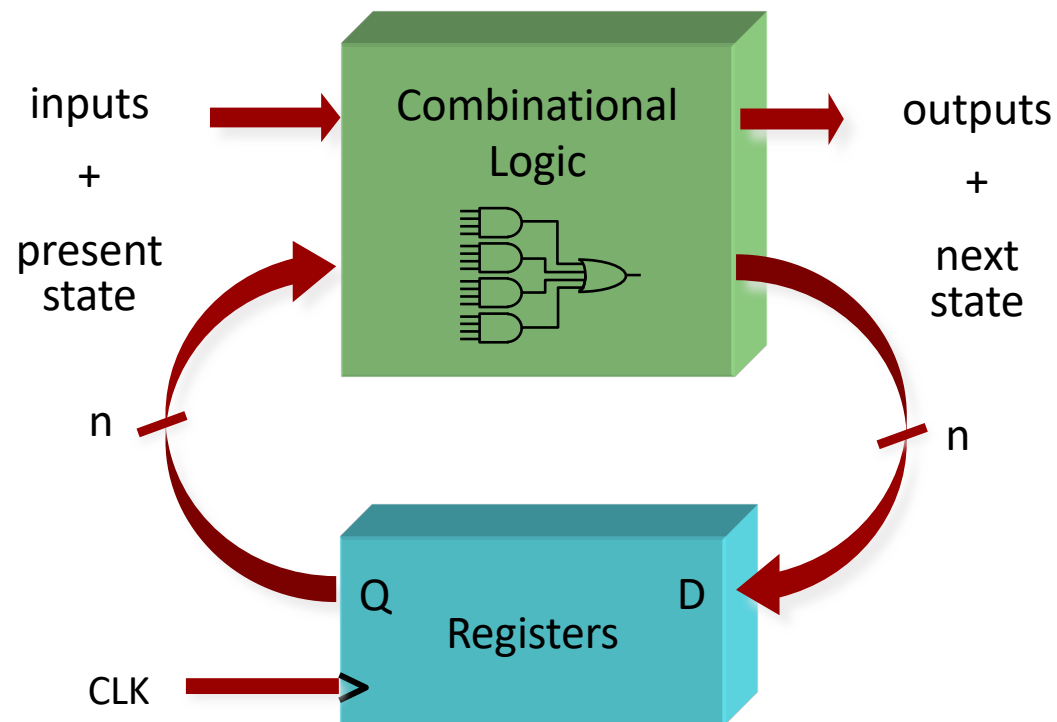
- One simple solution (~from Lab 02)
- One bit positive discrete time positive

```
module simple_soln( input wire clk,  
                   input wire l_in,  
                   output logic p_out);  
    logic old_l; //remember previous value!  
    always_ff @(posedge clk) begin  
        old_l <= l_in; //remember it!  
    end  
    assign p_out = l_in & ~ old_l; //high and prev low  
endmodule
```

- Let's try to formalize this a bit more

Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for **sequential circuits** with centralized “**states**” of operation
- At each clock edge, combinational logic computes **outputs** and **next state** as a function of **inputs** and **present state**



Level-to-Pulse

- **State:** how/what stores past information?
- **Output Logic:** How does state and input influence output
- **State Transition Logic:** Logic dictating next state
- **State Transition:** Actual updating of state

State

```
module simple_soln( input wire clk,
                    input wire l_in,
                    output logic p_out);
    logic old_l; //remember previous value!
    always_ff @(posedge clk) begin
        old_l <= l_in; //remember it!
    end
    assign p_out = l_in & ~old_l; //high and prev low
endmodule
```

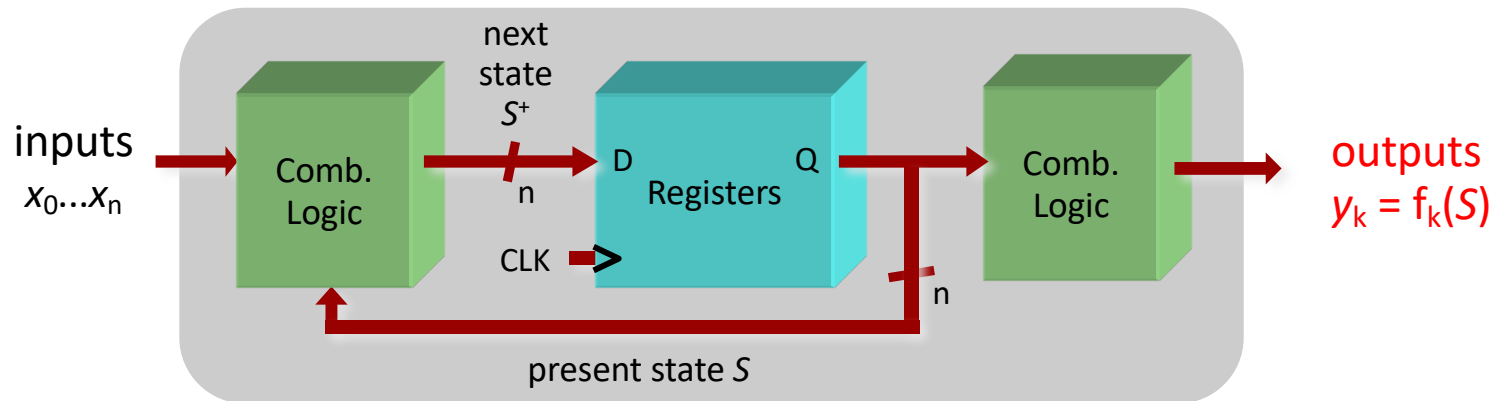
State Transition
and
State Transition Logic

Output Logic

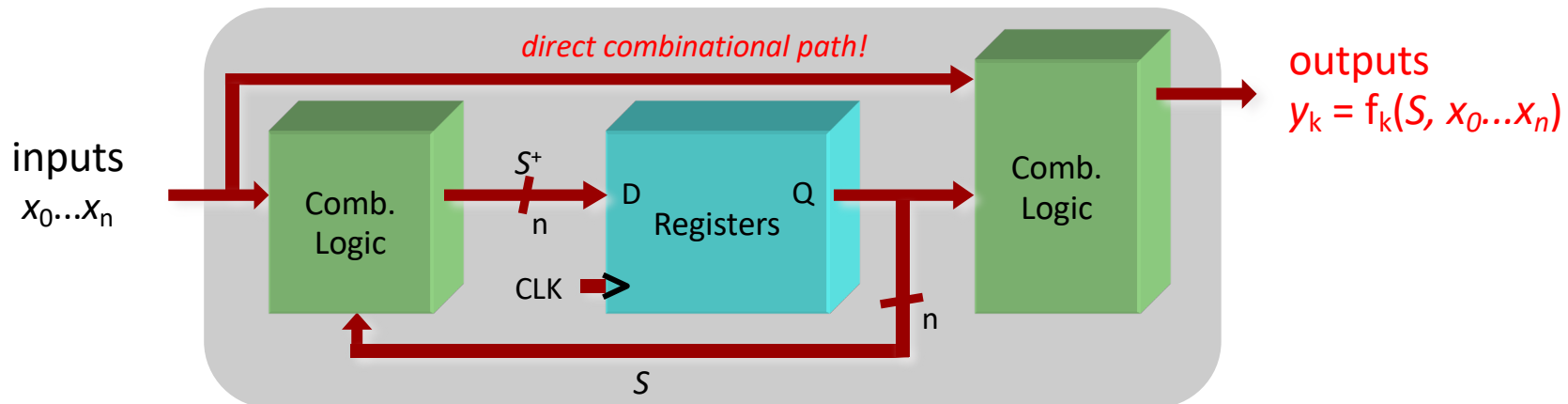
Let's Formalize it: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- **Moore FSM:**



- **Mealy FSM:**



Moore



- Edward F. Moore
- 1925-2003
- Virginia Tech
- Worked with Claude Shannon
- Not same Moore as Moore's Law...that was Gordon Moore from Intel

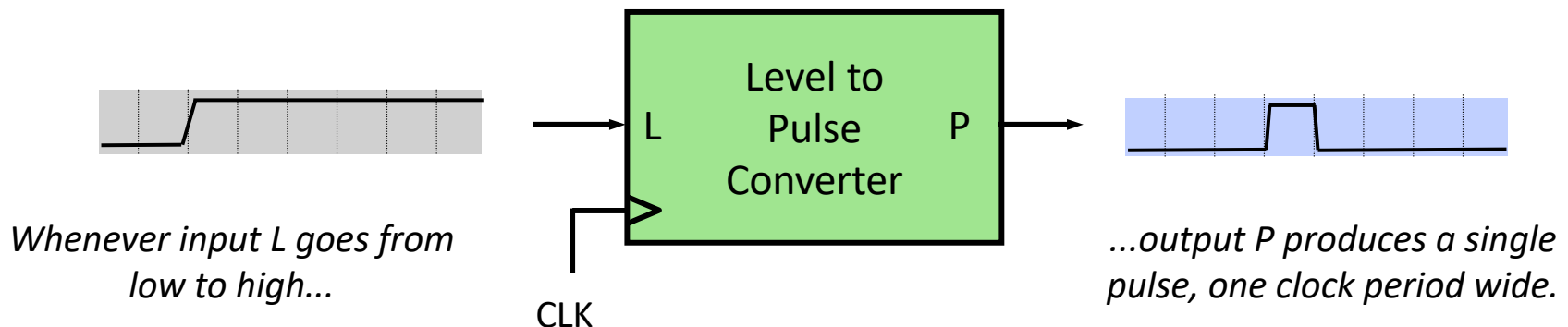
Mealy



- George H. Mealy
- 1927-2010
- Harvard, Bell Labs

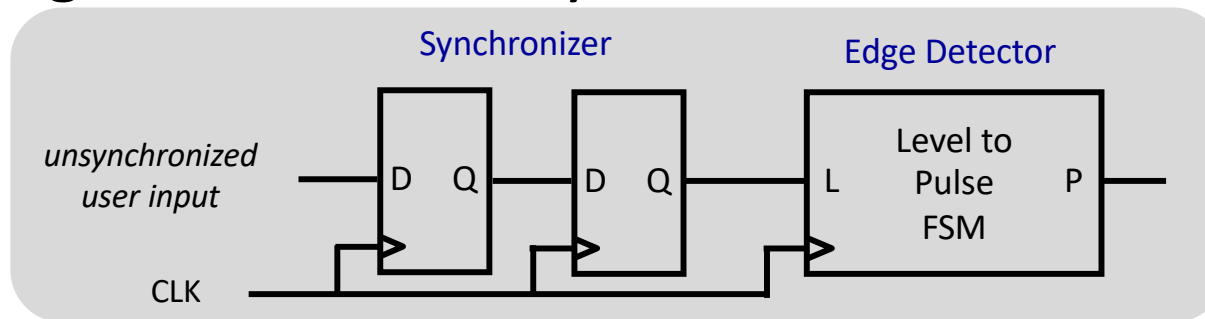
Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for modules

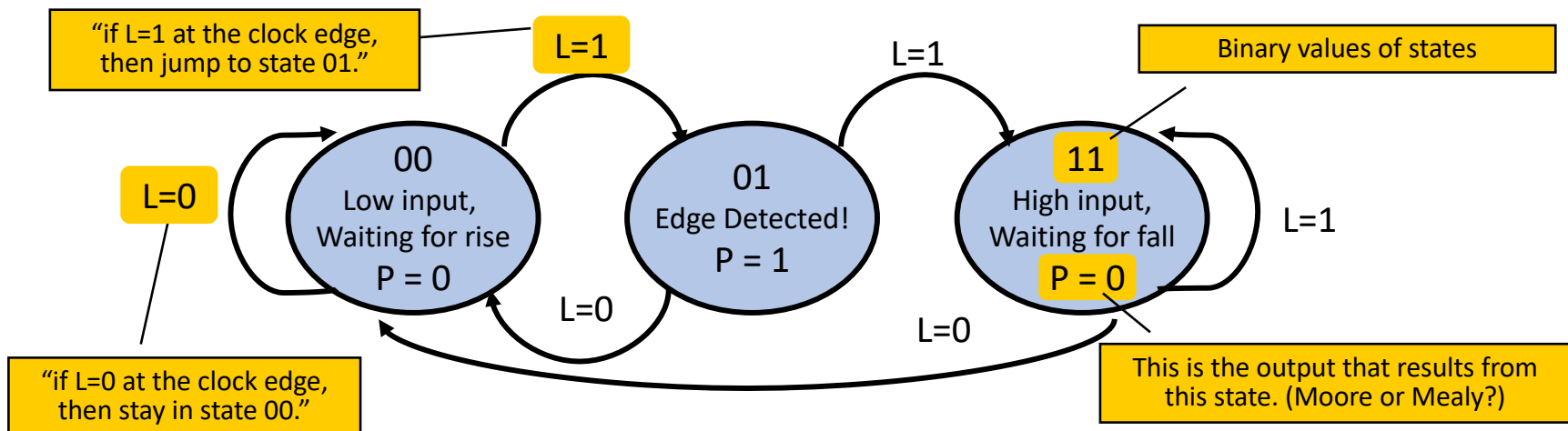


Step 1: State Transition Diagram

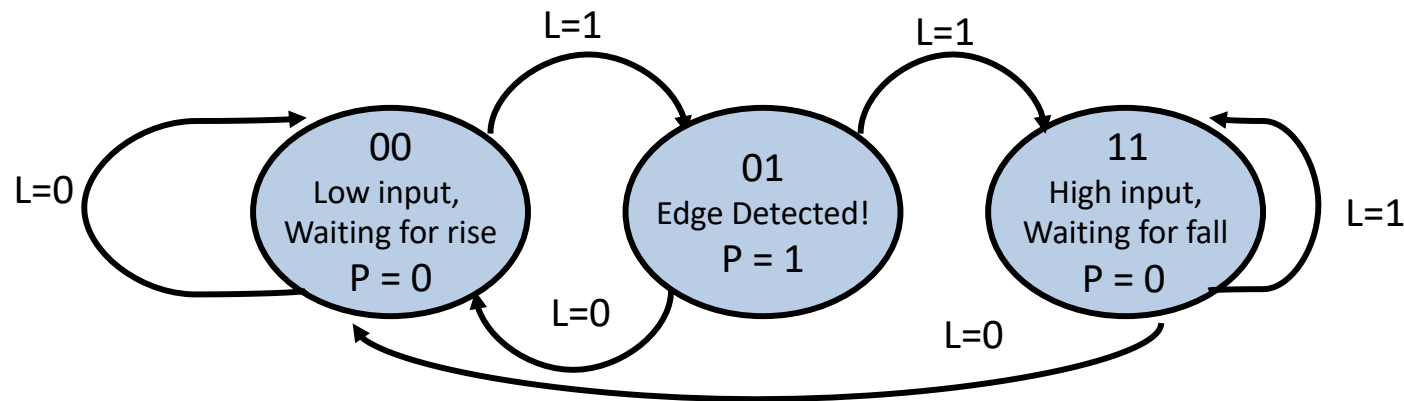
- Block diagram of desired system:



- State transition diagram** is a useful FSM representation and design aid:



Valid State Transition Diagrams



- Arcs leaving a state are **mutually exclusive**, i.e., for any combination input values there's at most one applicable arc
- Arcs leaving a state are **collectively exhaustive**, i.e., for any combination of input values there's at least one applicable arc**
- So for each state: for any combination of input values there's exactly one applicable arc (**no ambiguity**)
- Often a starting state is specified
- Each state specifies values for all outputs (in the case of Moore)

Choosing State Representation

Choice #1: binary encoding

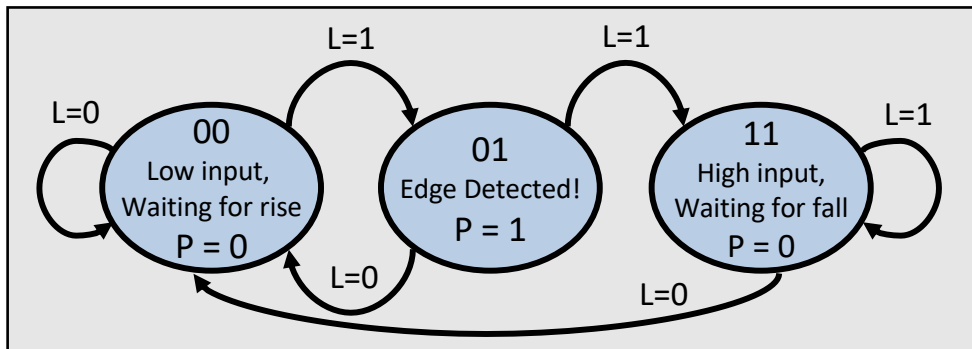
For N states, use $\lceil \log_2 N \rceil$ bits to encode the state with each state represented by a unique combination of the bits. Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.

Choice #2: “one-hot” encoding

For N states, use N bits to encode the state where the bit corresponding to the current state is 1, all the others 0. Tradeoffs: more state registers, but often much less combinational logic since state decoding is trivial.

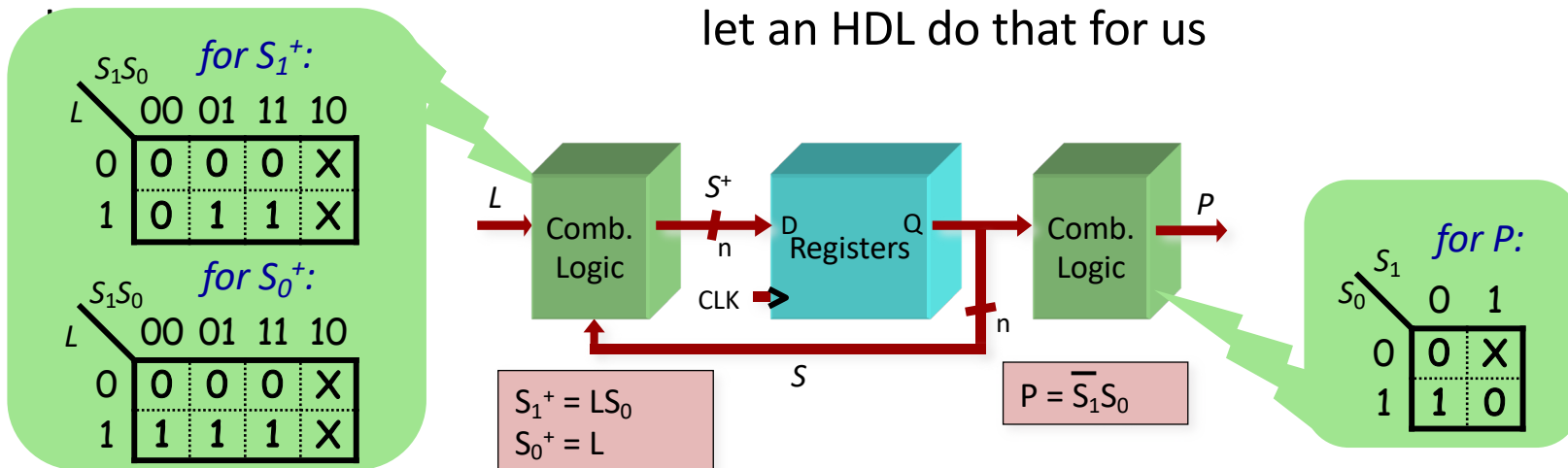
Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)

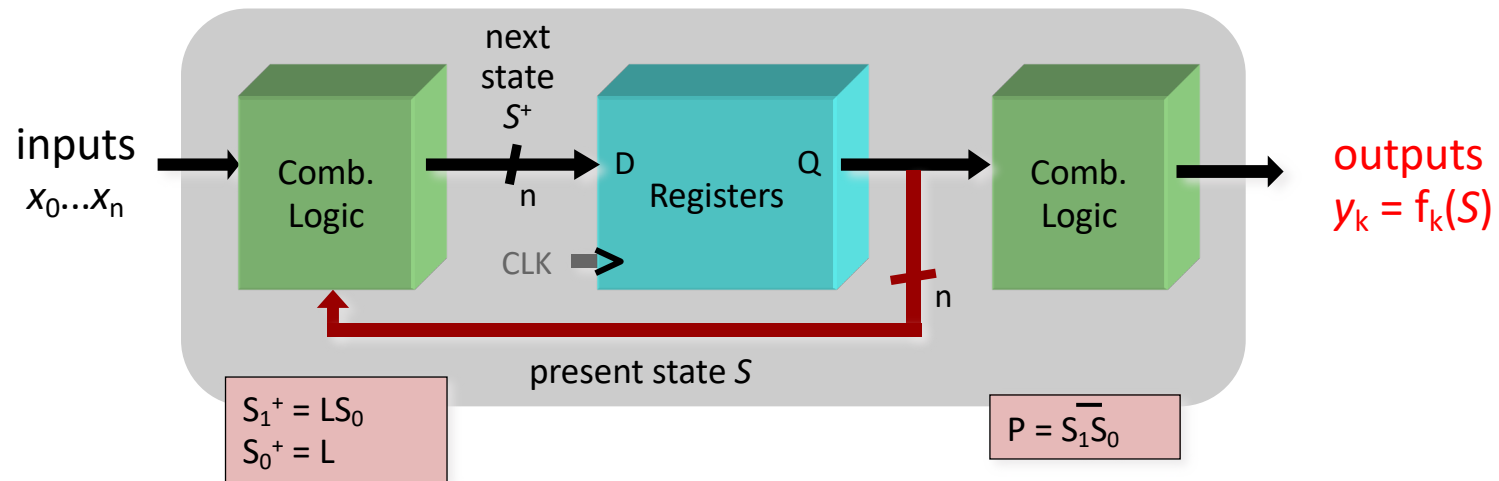


Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

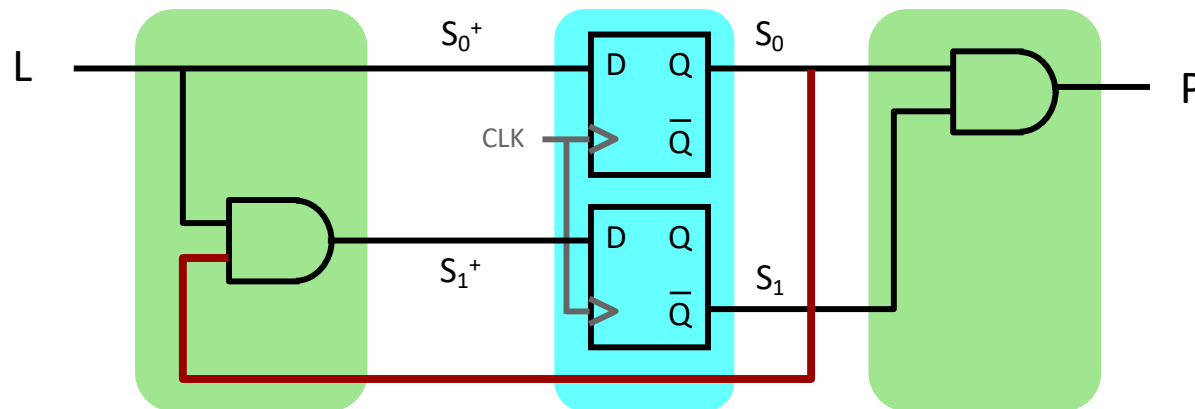
- Combinational logic **could** be derived using Karnaugh maps by hand, let an HDL do that for us



Moore Level-to-Pulse Converter



Moore FSM circuit implementation of level-to-pulse converter:



Moore Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Moore FSM implementation of the level-to-pulse converter:

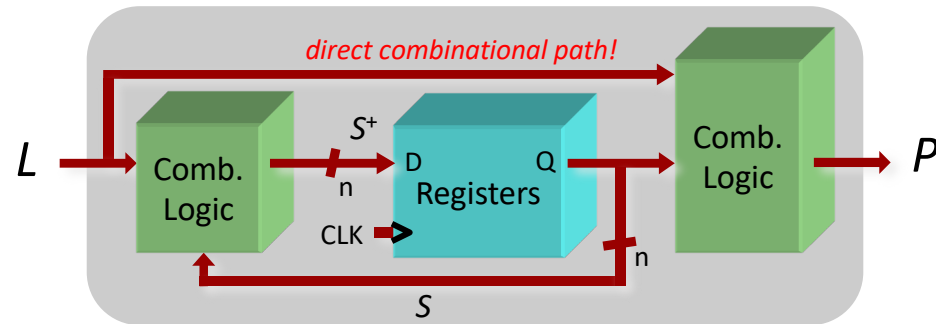
```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0; //define your states as...
    localparam EDGE_DETECTED = 2'b01; //parameters for easy...
    localparam HIGH_WAITING = 2'b10; //reading!
    logic [1:0] state; //contain state!
    logic [1:0] next_state; //hold next state!
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0; //output based only on...
            EDGE_DETECTED: p_out = 1'b1; //current state! This is...
            HIGH_WAITING: p_out = 1'b0; //a characteristic of Moore FSM
            default: p_out = 1'b0;
        endcase
    end
    //State Transition Logic (Combinational):
    always_comb begin
        case(state) //Also consider explicit if/elses
            LOW_WAITING: next_state = l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        state <= next_state; //state becomes calculated next_state
    end
endmodule
```

Moore Level-to-Pulse Converter (SystemVerilog)

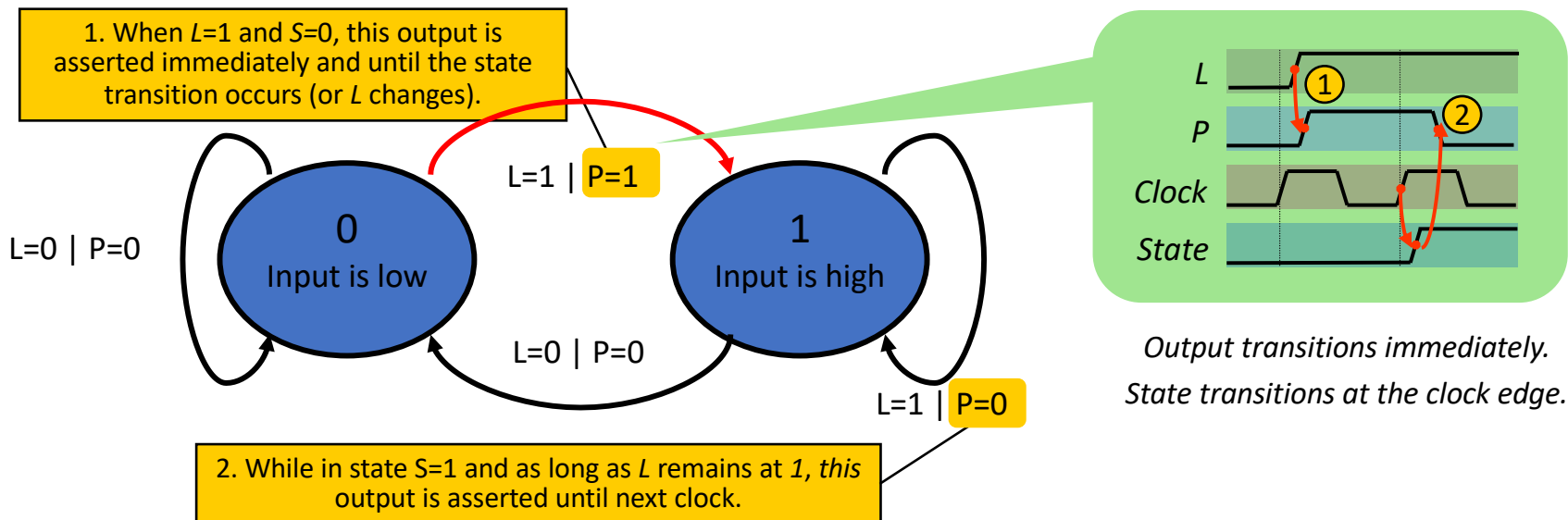
- Merging **State Transition Logic** and **State Transition** into one block
- Some people like this more

```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0;
    localparam EDGE_DETECTED = 2'b01;
    localparam HIGH_WAITING = 2'b10;
    logic [1:0] state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0;
            EDGE_DETECTED: p_out = 1'b1;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Logic:
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

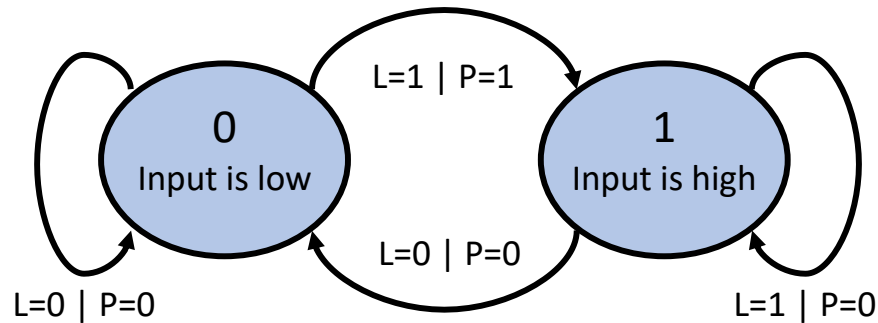
Design of a Mealy Level-to-Pulse



- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations

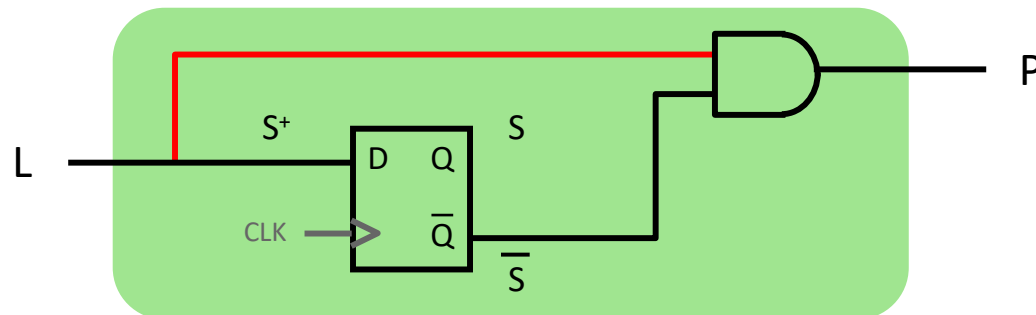


Mealy Level-to-Pulse Converter



Pres. State	In	Next State	Out
S	L	S^+	P
0	0	0	0
0	1	1	1
1	1	1	0
1	0	0	0

Mealy FSM circuit implementation of level-to-pulse converter:



- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

Mealy Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Mealy FSM implementation of the level-to-pulse converter:

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0; //define states but notice...
    localparam HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
    //though not always, is like that
    logic state; //state (smaller than before...only two states to rep)
    logic next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition Logic:
    always_comb begin
        case(state)
            LOW_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well (same goes for any...
        //clocked logic block)
        state <= next_state;
    end
endmodule
```

Mealy Level-to-Pulse Converter (SystemVerilog)

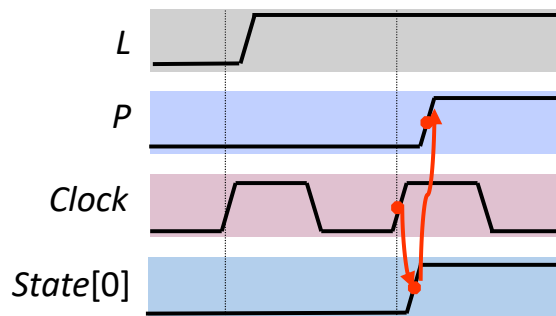
- Merging **State Transition Logic** and **State Transition** into one block

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0;
    localparam HIGH_WAITING = 1'b1;
    logic state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Transition Logic!
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

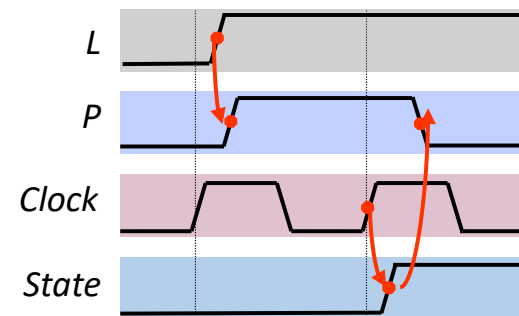
Moore/Mealy Trade-Offs

- How are they different?
 - Moore: **outputs = f(state)** only
 - Mealy **outputs = f(state *and* input)**
 - Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P



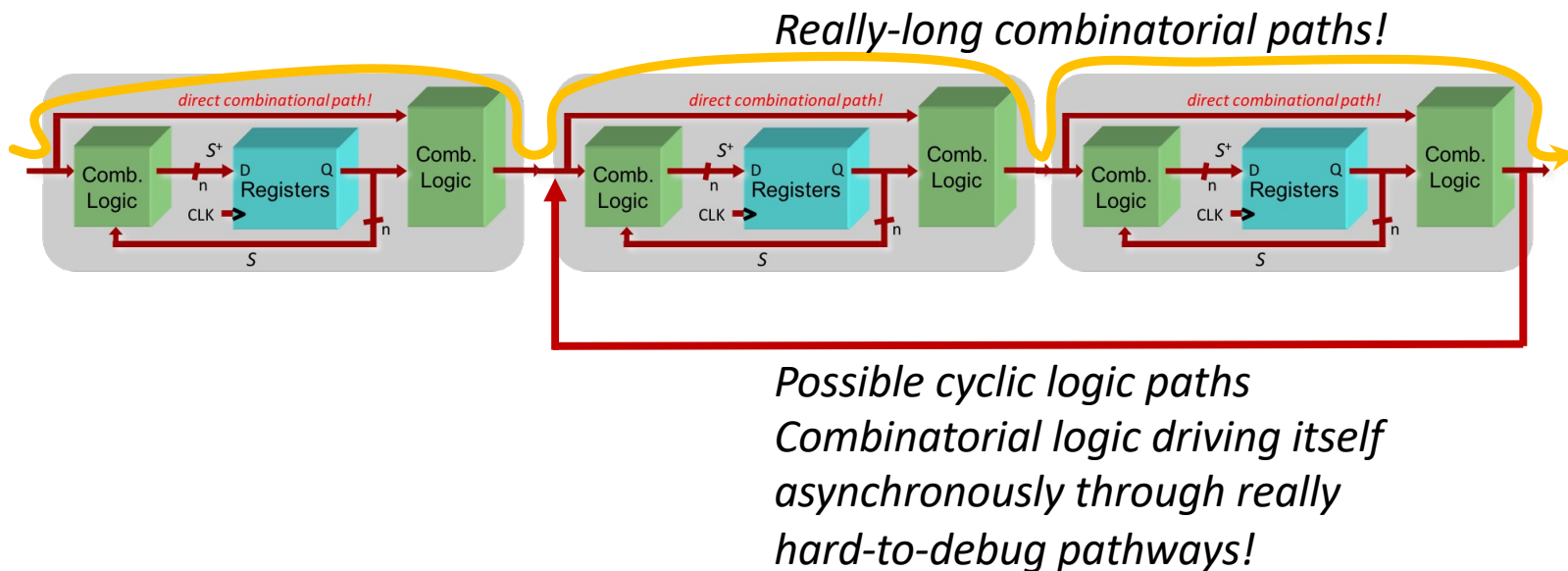
Mealy: immediate assertion of P



- Compared to a Moore FSM, a Mealy FSM ***might***...
 - Be more difficult to conceptualize and design (both at circuit level and in HDL)
 - Have fewer states
 - Be expressed using fewer lines of Verilog

Moore/Mealy Trade-Offs

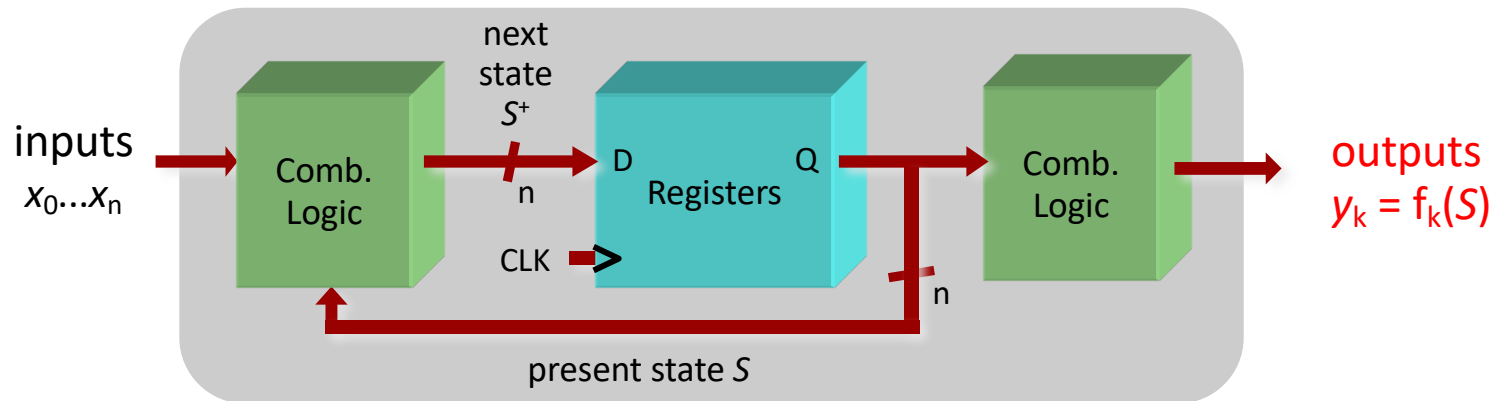
- Moore:
 - Usually more states
 - Each state has a particular output
- Mealy:
 - Fewer states, outputs are specified on edges of diagram
 - Potential Dangers:



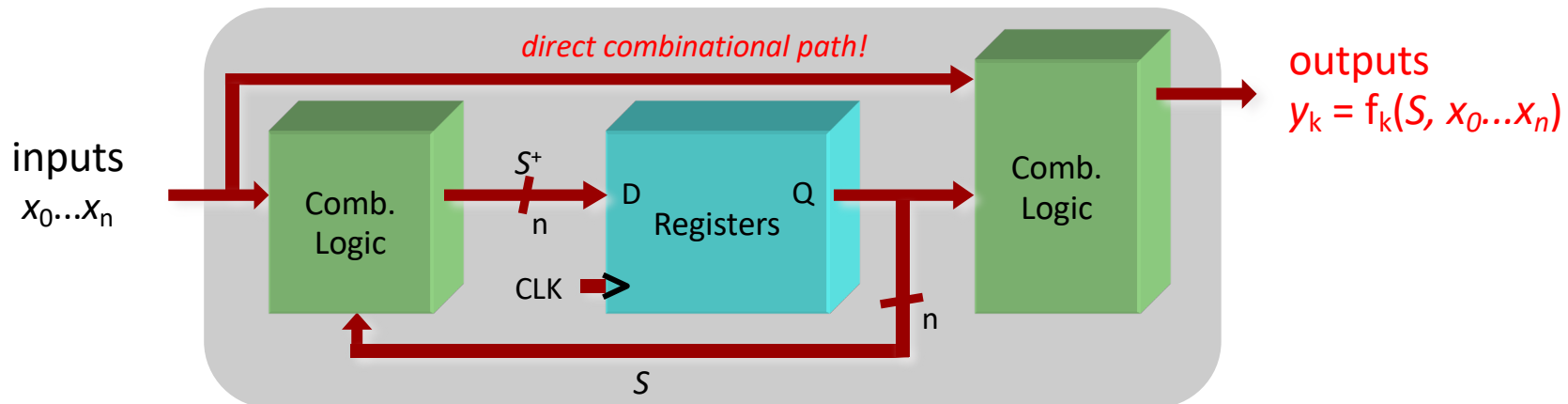
So Review: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- Moore FSM:



- Mealy FSM:



FSM Examples

Time-Dependent

FSM Example

GOAL:

- Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output signal. The combination will **always be 01011**.
- We will encode the lock into the state.
- Use a sliding window of the last five entries

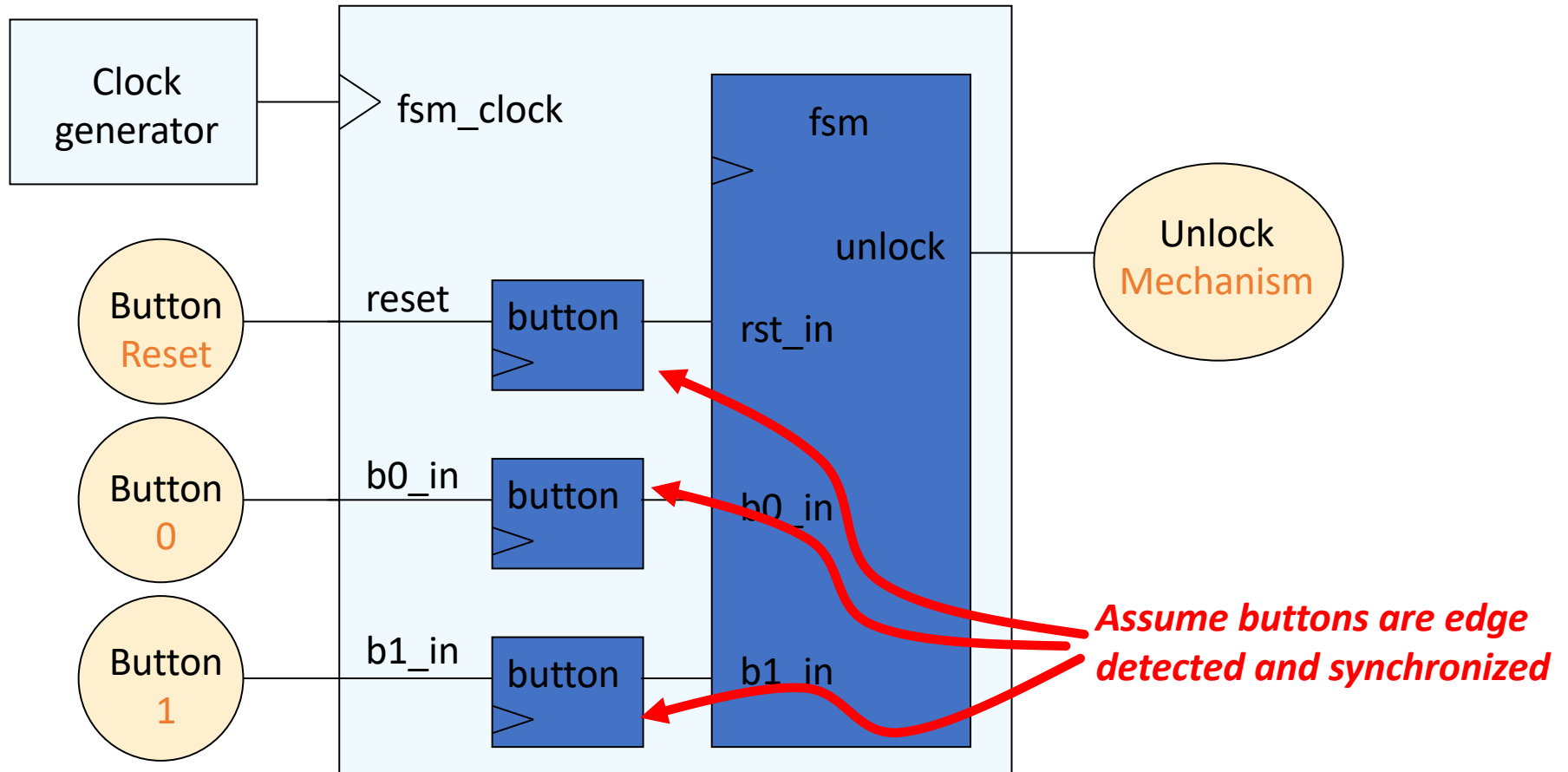


STEPS:

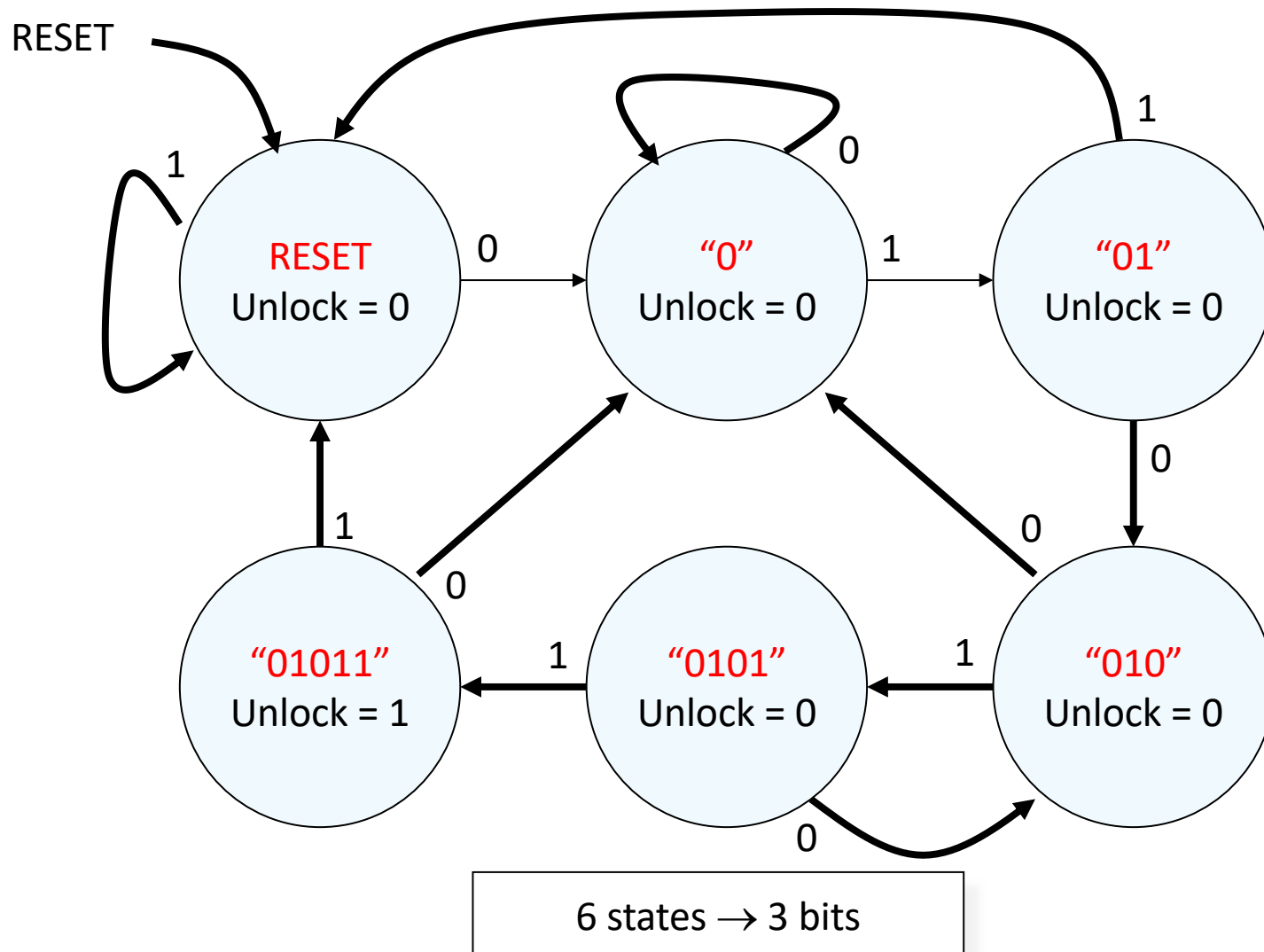
1. Design lock FSM (block diagram, state transitions)
2. Write SystemVerilog module(s) for FSM

Step 1A: Block Diagram

lock



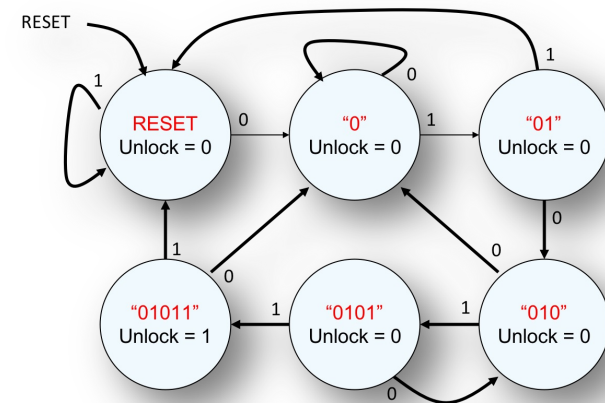
Step 1B: State transition diagram



Step 2: Write Verilog

```
module lock( input wire clk,rst,b0_in,b1_in,
             output logic unlock_out);
    // implement state transition diagram
    logic [2:0] state,next_state;
    always_comb begin
        // combinational logic!
        next_state = ???;
    end
    always_ff @(posedge clk) begin
        state <= next_state;
    end
    // generate output
    assign out = ???;
endmodule
```

Step 2B: state transition diagram



```

localparam S_RESET = 0; parameter S_0 = 1; // state assignments
localparam S_01 = 2; parameter S_010 = 3;
localparam S_0101 = 4; parameter S_01011 = 5;
logic [2:0] state, next_state; //(both 3 bits wide)
always_comb begin // implement state transition diagram
    if (rst) next_state = S_RESET;
    else begin
        case (state)
            S_RESET: next_state = b0_in ? S_0 : b1_in ? S_RESET : state;
            S_0: next_state = b0_in ? S_0 : b1_in ? S_01 : state;
            S_01: next_state = b0_in ? S_010 : b1_in ? S_RESET : state;
            S_010: next_state = b0_in ? S_0 : b1_in ? S_0101 : state;
            S_0101: next_state = b0_in ? S_010 : b1_in ? S_01011 : state;
            S_01011: next_state = b0_in ? S_0 : b1_in ? S_RESET : state;
            default: next_state = S_RESET; // handle unused states
        endcase
    end
end

```

Step 2C: generate output

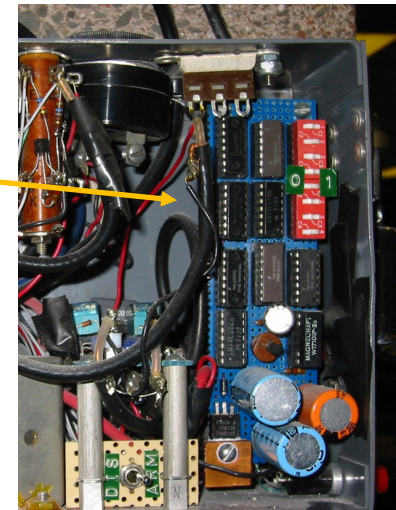
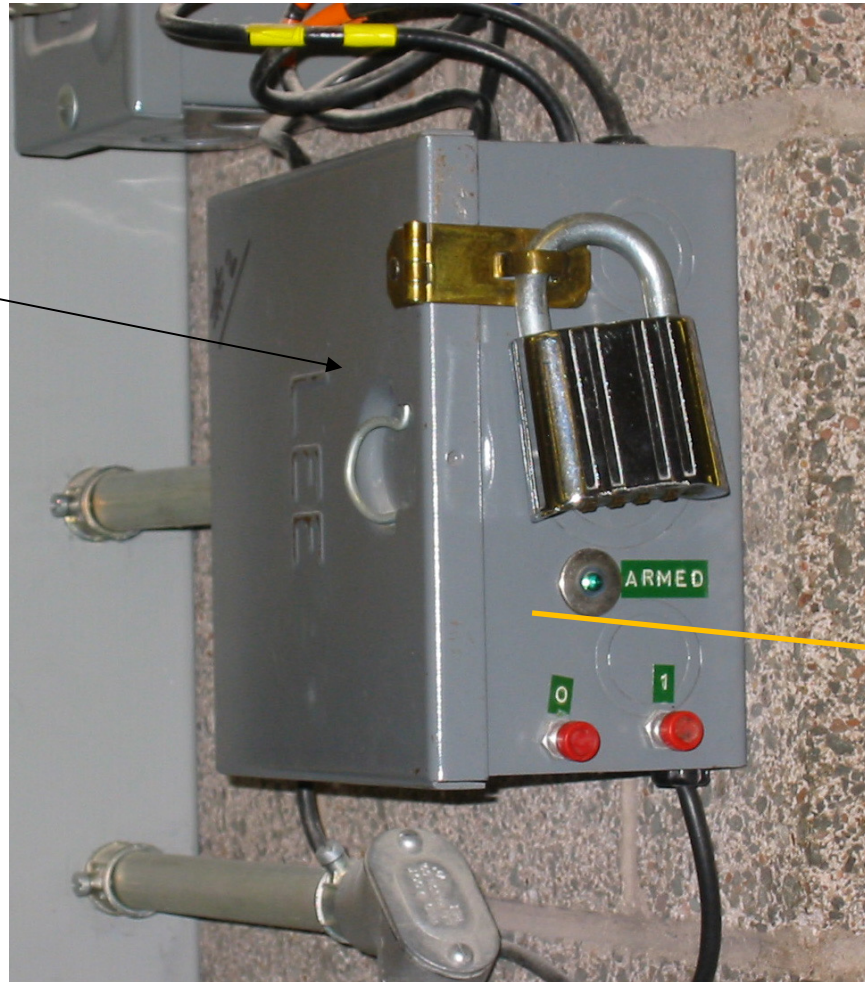
// it's a Moore machine! Output only depends on current state

```
assign unlock_out = (state == S_01011); // assign output: Moore machine
```

Step 2: final Verilog implementation

```
module lock(input wire clk_in,rst_in,b0_in,b1_in,
            output logic unlock_out);
    localparam S_RESET = 0; parameter S_0 = 1; // state assignments
    localparam S_01 = 2; parameter S_010 = 3;
    localparam S_0101 = 4; parameter S_01011 = 5;
    logic [2:0] state, next_state; //(both 3 bits wide)
    always_comb begin // implement state transition diagram
        if (rst_in) next_state = S_RESET;
        else begin
            case (state)
                S_RESET: next_state = b0_in ? S_0 : b1_in ? S_RESET : state;
                S_0: next_state = b0_in ? S_0 : b1_in ? S_01 : state;
                S_01: next_state = b0_in ? S_010 : b1_in ? S_RESET : state;
                S_010: next_state = b0_in ? S_0 : b1_in ? S_0101 : state;
                S_0101: next_state = b0_in ? S_010 : b1_in ? S_01011 : state;
                S_01011: next_state = b0_in ? S_0 : b1_in ? S_RESET : state;
                default: next_state = S_RESET; // handle unused states
            endcase
        end
    end
    always_ff @(posedge clk) state <= next_state;
    assign unlock_out = (state == S_01011); // assign output: Moore machine
endmodule
```

Real FSM Security System



The 6.111 Vending Machine

(example from circa 2000...slightly updated)

- Lab assistants demand a new soda machine for the 6.111 lab. You design the FSM controller.
- **All selections are \$0.30.**
- The machine makes change. (Dimes and nickels only.)
- Inputs: limit 1 per clock
 - Q - quarter inserted
 - D - dime inserted
 - N - nickel inserted
- Outputs: limit 1 per clock
 - DC - dispense can
 - DD - dispense dime
 - DN - dispense nickel



What States are in the System?

- A starting (idle) state:



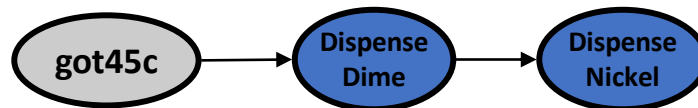
- A state for each possible amount of money captured:



- What's the maximum amount of money captured before purchase?
25 cents (just shy of a purchase) + one quarter (largest coin)

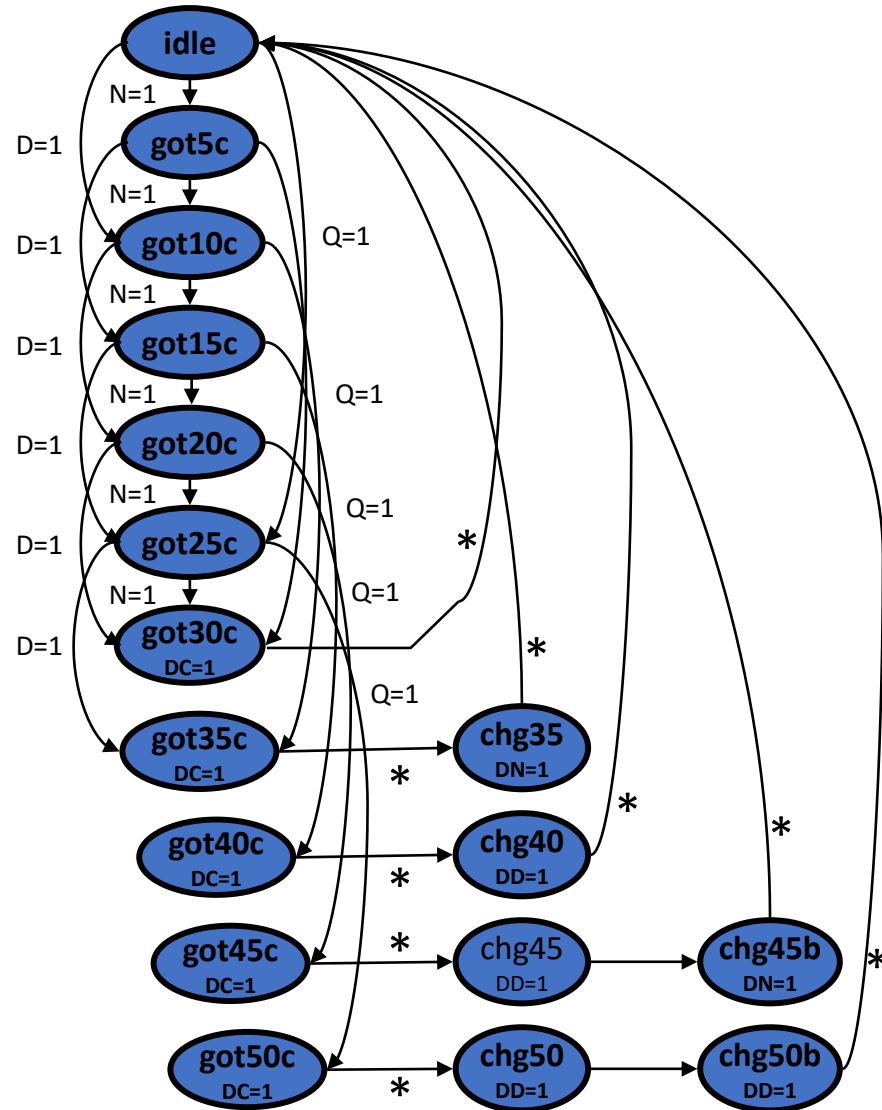


- States to dispense change (one per coin dispensed):



A Moore Vender

Here's a first cut at the state transition diagram.

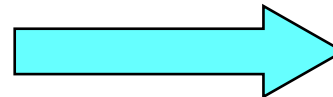
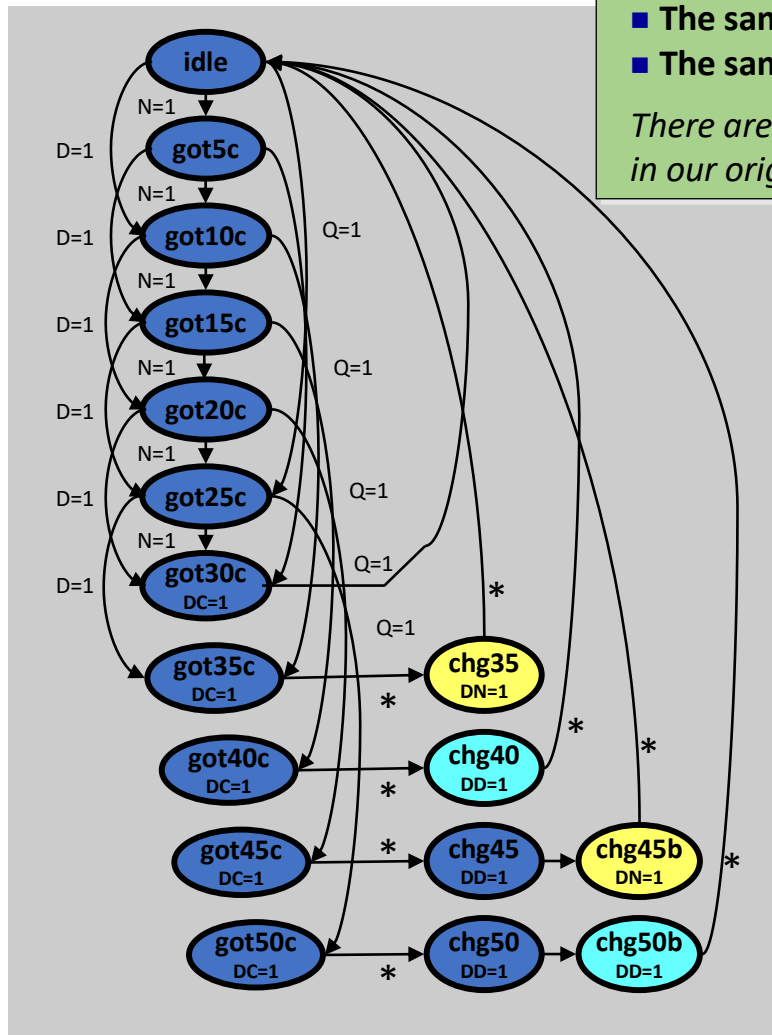


State Reduction

Duplicate states have:

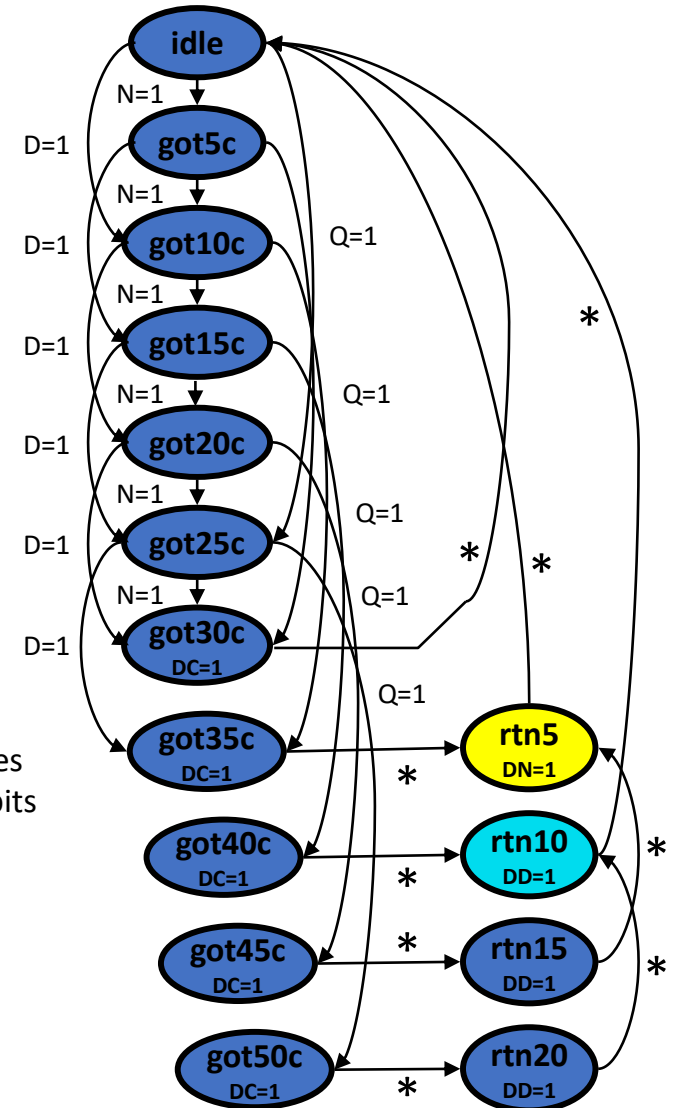
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.



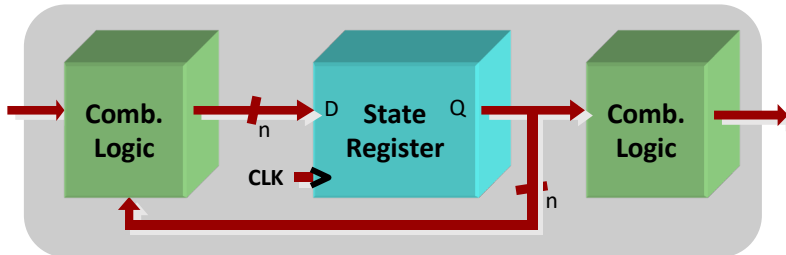
17 states
5 state bits

15 states
4 state bits



Verilog for the Moore Vender

States defined with
parameter keyword



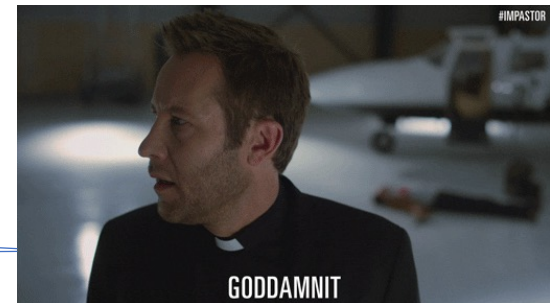
- **State register**
(sequential always block) *So triggered on posedge clock*
- **Next-state combinational logic**
(comb. always block with case)
- **Output combinational logic block**
(comb. always block or assign statements)

State register defined with
sequential always block
(always_ff)

```
module mooreVender (  
    input wire N, D, Q, clk, reset,  
    output logic DC, DN, DD,  
    output logic [3:0] state);  
    logic [3:0] next;  
  
    parameter IDLE = 0;  
    parameter GOT_5c = 1;  
    parameter GOT_10c = 2;  
    parameter GOT_15c = 3;  
    parameter GOT_20c = 4;  
    parameter GOT_25c = 5;  
    parameter GOT_30c = 6;  
    parameter GOT_35c = 7;  
    parameter GOT_40c = 7;  
    parameter GOT_45c = 9;  
    parameter GOT_50c = 10;  
    parameter RETURN_20c = 11;  
    parameter RETURN_15c = 12;  
    parameter RETURN_10c = 13;  
    parameter RETURN_5c = 14;  
  
    always_ff @(posedge clk) begin  
        if (!reset) state <= IDLE;  
        else state <= next;  
    end
```

Enums in SystemVerilog

```
module mooreVender (  
    input wire N, D, Q, clk, reset,  
    output logic DC, DN, DD,  
    output logic [3:0] state);  
    logic [3:0] next;  
  
    parameter IDLE = 0;  
    parameter GOT_5c = 1;  
    parameter GOT_10c = 2;  
    parameter GOT_15c = 3;  
    parameter GOT_20c = 4;  
    parameter GOT_25c = 5;  
    parameter GOT_30c = 6;  
    parameter GOT_35c = 7;  
    parameter GOT_40c = 7;  
    parameter GOT_45c = 9;  
    parameter GOT_50c = 10;  
    parameter RETURN_20c = 11;  
    parameter RETURN_15c = 12;  
    parameter RETURN_10c = 13;  
    parameter RETURN_5c = 14;  
  
    always_ff @(posedge clk) begin  
        if (!reset) state <= IDLE;  
        else state <= next;  
    end
```



Same value...uh oh

Enums in SystemVerilog

```
module mooreVender (
  input wire N, D, Q, clk, reset,
  output logic DC, DN, DD,
  output logic [3:0] state_out);

  enum {IDLE, GOT_5c, GOT_10c, GOT_15c, GOT_20c,
        GOT_25c, GOT_35c, GOT_40c, GOT_45c,
        GOT_50c, RETURN_20c, RETURN_15c,
        RETURN_10c, RETURN_5c } state, next;

  assign state_out = state;
  always_ff @(posedge clk or negedge reset) begin
    if (!reset) state <= IDLE;
    else state <= next;
  end
```

- State and next_state are now restricted to only be one of a set of values
- Vivado figures out the most efficient encoding
- Ensures you don't make duplicates or do other stupid mistakes

```
typedef enum {IDLE, GOT_5c, GOT_10c, GOT_15c, GOT_20c,
              GOT_25c, GOT_35c, GOT_40c, GOT_45c,
              GOT_50c, RETURN_20c, RETURN_15c,
              RETURN_10c, RETURN_5c } coin_state;

coin_state state, next; //instances here
```

Next-state logic within a **combinational always** block

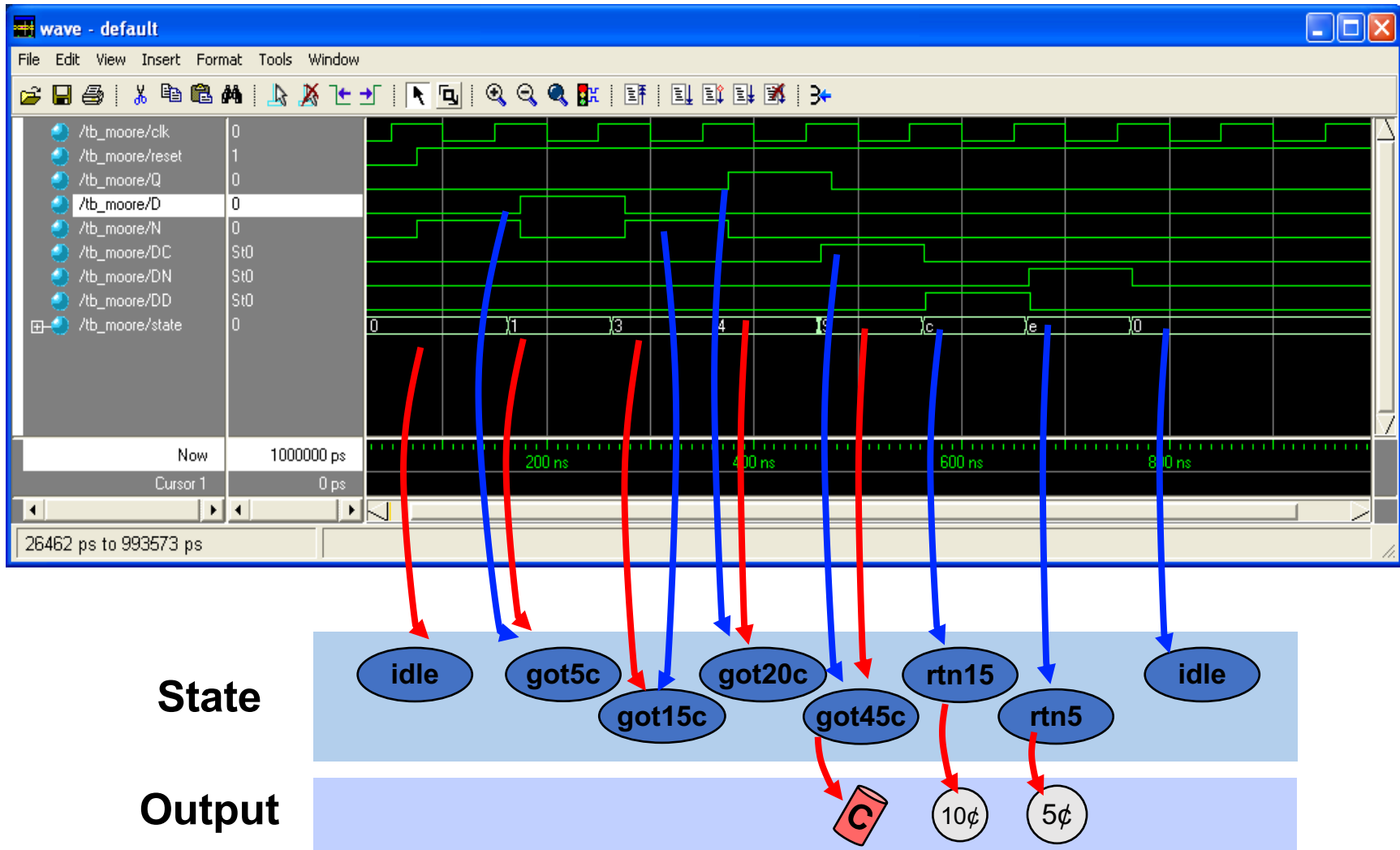
```
always_comb (state or N or D or Q) begin
  case (state)
    IDLE: if (Q) next = GOT_25c;
          else if (D) next = GOT_10c;
          else if (N) next = GOT_5c;
          else next = IDLE;
    GOT_5c: if (Q) next = GOT_30c;
            else if (D) next = GOT_15c;
            else if (N) next = GOT_10c;
            else next = GOT_5c;
    GOT_10c: if (Q) next = GOT_35c;
             else if (D) next = GOT_20c;
             else if (N) next = GOT_15c;
             else next = GOT_10c;
    GOT_15c: if (Q) next = GOT_40c;
             else if (D) next = GOT_25c;
             else if (N) next = GOT_20c;
             else next = GOT_15c;
    GOT_20c: if (Q) next = GOT_45c;
             else if (D) next = GOT_30c;
             else if (N) next = GOT_25c;
             else next = GOT_20c;
    GOT_25c: if (Q) next = GOT_50c;
             else if (D) next = GOT_35c;
             else if (N) next = GOT_30c;
             else next = GOT_25c;
    GOT_30c: next = IDLE;
    GOT_35c: next = RETURN_5c;
    GOT_40c: next = RETURN_10c;
    GOT_45c: next = RETURN_15c;
    GOT_50c: next = RETURN_20c;
    RETURN_20c: next = RETURN_10c;
    RETURN_15c: next = RETURN_5c;
    RETURN_10c: next = IDLE;
    RETURN_5c: next = IDLE;
    default: next = IDLE;
  endcase
end
```

Verilog for the Moore Vender

Combinational output assignment

```
assign DC = (state == GOT_30c ||
             state == GOT_35c ||
             state == GOT_40c ||
             state == GOT_45c ||
             state == GOT_50c);
assign DN = (state == RETURN_5c);
assign DD = (state == RETURN_20c ||
             state == RETURN_15c ||
             state == RETURN_10c);
endmodule
```

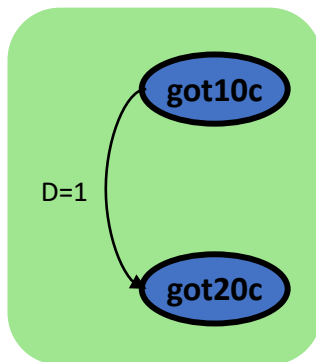
Simulation of Moore Vender



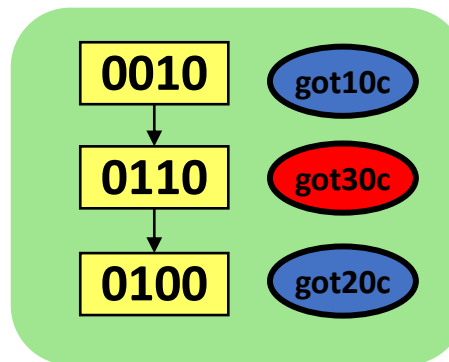
FSM Output Glitching

- FSM state bits may not transition at precisely the same time
- Combinational logic for outputs may contain hazards/glitches
- Result: your FSM outputs may glitch!

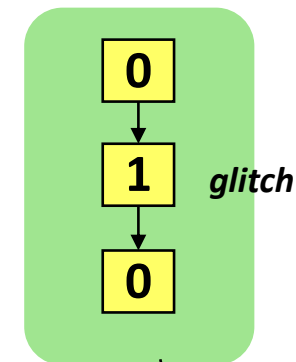
during this state transition...



...the state registers may transition like this...



...causing the DC output to **glitch** like this!



```
assign DC = (state == GOT_30c ||  
             state == GOT_35c ||  
             state == GOT_40c ||  
             state == GOT_45c ||  
             state == GOT_50c);
```

If the soda dispenser is glitch-sensitive, your customers can get a 20-cent soda!

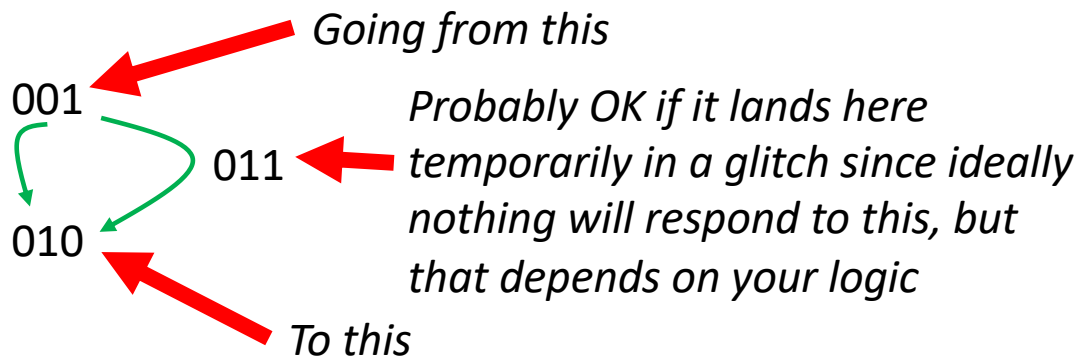
One way to fix Glitches:

- Don't have to have state 3 (3'b011) go into state 4 (3'b100). Use different state naming/use different numbers!!! *A rose by any other name would smell as sweet*
- Perhaps a Gray code (??):
 - Count up like: 000, 001, 011, 010, 110, 111, 101, 100, ...
 - Have the really important/glitch-sensitive states only require transitions of one bit

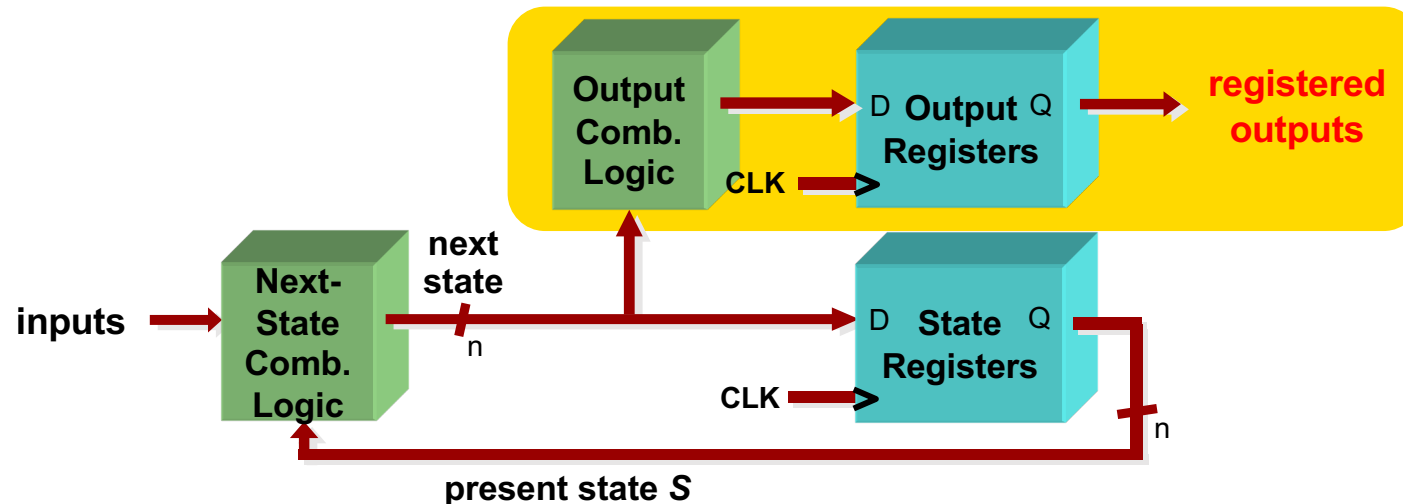
- One-hot encoding:

- Johnson encoding:

000
100
110
111
011
001



Another Solution: Registered FSM Outputs are Glitch-Free



- Move output generation into the sequential always block
- Calculate outputs based on next state
- Delays outputs by one clock cycle. Problematic in some application.

```
always_ff @(posedge clk) begin
    if (!reset) begin
        state <= IDLE;
    end else begin
        state <= next;
        DC <= (next == GOT_30c || next == GOT_35c ||
               next == GOT_40c || next == GOT_45c ||
               next == GOT_50c);
        DN <= (next == RETURN_5c);
        DD <= (next == RETURN_20c || next == RETURN_15c ||
               next == RETURN_10c);
    end
```

*Note this is inside an edged always with non-blocking assigns!
This will synthesize to registered outputs!*

Encoding with Enums?

- Generally in SystemVerilog, an enum, unless specified will be 0, 1, 2, 3, etc...

```
enum {IDLE, GOT_5c, GOT_10c, GOT_15c, GOT_20c,  
      GOT_25c, GOT_35c, GOT_40c, GOT_45c,  
      GOT_50c, RETURN_20c, RETURN_15c,  
      RETURN_10c, RETURN_5c } state, next;
```

- When synthesizing, Vivado may decide on a different encoding, however!
- May or may not be what you want! Or you want a particular encoding

```
typedef enum {IDLE, GOT_5c, GOT_10c, GOT_15c, GOT_20c,  
              GOT_25c, GOT_35c, GOT_40c, GOT_45c,  
              GOT_50c, RETURN_20c, RETURN_15c,  
              RETURN_10c, RETURN_5c } state_enc_T;  
  
state_enc_T state, next;
```

Encoding in Enums?

- If want one-hot do:

```
typedef enum {IDLE, GOT_5c, GOT_10c, GOT_15c, GOT_20c,  
             GOT_25c, GOT_35c, GOT_40c, GOT_45c,  
             GOT_50c, RETURN_20c, RETURN_15c,  
             RETURN_10c, RETURN_5c } state_enc_T;  
  
(* fsm_encoding = "one_hot" *) state_enc_T state, next;
```

- Can also do specify sequential, johnson, gray encoding, etc... or you can specify your own if you have a good idea:

Division (an example of an algorithm that takes an unknown amount of time)

```
def divider (dividend, divisor):  
    count = 0  
    if divisor==0:  
        return -1  
    while dividend>=divisor:  
        dividend -= divisor  
        count += 1  
    return (count, dividend)
```

Super efficient divider \s

A Divider

- This is a Verilog FSM example of the algorithm on the previous page which will run an unknown number of times given a set of inputs
- This is how the functionality of a while loop could be developed in your modules
- Will not handle negative, or 0 or other things...

Code on the site's lecture page

```
3
4 module divider #(parameter WIDTH = 32)
5 (
6     input wire clk,
7     input wire rst,
8     input wire[WIDTH-1:0] dividend,
9     input wire[WIDTH-1:0] divisor,
10    input wire data_in_valid,
11    output logic[WIDTH-1:0] quotient,
12    output logic[WIDTH-1:0] remainder,
13    output logic data_out_valid,
14    output logic error,
15    output logic busy
16 );
17 logic [WIDTH-1:0] quotient_g;
18 logic [WIDTH-1:0] dividend_h;
19 logic [WIDTH-1:0] divisor_h;
20 enum {RESTING, DIVIDING} state;
21 always_ff @(posedge clk)begin
22     if (rst)begin
23         quotient_g <= 0;
24         dividend_h <= 0;
25         divisor_h <= 0;
26         remainder <= 0;
27         quotient <= 0;
28         busy <= 1'b0;
29         error <= 1'b0;
30         state <= RESTING;
31         data_out_valid <= 1'b0;
32     end else begin
33         case (state)
34             RESTING: begin
35                 if (data_in_valid)begin
36                     state <= DIVIDING;
37                     quotient_g <= 0;
38                     dividend_h <= dividend;
39                     divisor_h <= divisor;
40                     busy <= 1'b1;
41                     error <= 1'b0;
42                     data_out_valid <= 1'b0;
43                 end
44             DIVIDING: begin
45                 if (dividend_h <= 0)begin
46                     state <= RESTING; //similar to return statement
47                     remainder <= dividend_h;
48                     quotient <= quotient_g;
49                     busy <= 1'b0; //tell outside world i'm done
50                     error <= 1'b0;
51                     data_out_valid <= 1'b1; //good stuff!
52                 end else if (divisor_h == 0)begin
53                     state <= RESTING;
54                     remainder <= 0;
55                     quotient <= 0;
56                     busy <= 1'b0; //tell outside world i'm done
57                     error <= 1'b1; //ERROR
58                     data_out_valid <= 1'b1; //valid ERROR
59                 end else if (dividend_h < divisor_h) begin
60                     state <= RESTING;
61                     remainder <= dividend_h;
62                     quotient <= quotient_g;
63                     busy <= 1'b0;
64                     error <= 1'b0;
65                     data_out_valid <= 1'b1; //good stuff!
66                 end else begin
67                     //state staying in.
68                     state <= DIVIDING;
69                     quotient_g <= quotient_g + 1'b1;
70                     dividend_h <= dividend_h - divisor_h;
71                 end
72             end
73         endcase
74     end
75 end
76 endmodule
77
78 `default_nettype wire
79
```